UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

SERVIÇO DE COORDENAÇÃO PARA BASES DE DADOS REPLICADAS

Rui Jorge Raposo Posse

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2011

UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

SERVIÇO DE COORDENAÇÃO PARA BASES DE DADOS REPLICADAS

Rui Jorge Raposo Posse

DISSERTAÇÃO

Projecto orientado pelo Prof. Doutor Alysson Neves Bessani

MESTRADO EM ENGENHARIA INFORMÁTICA
Especialização em Arquitectura, Sistemas e Redes de Computadores

2011

Agradecimentos

Em primeiro lugar, quero deixar uma palavra de agradecimento à minha família. Aos meus pais, por me terem dado todas as condições para hoje ser quem sou e, se sou algo de bom, é devido a todos os princípios e valores com que eles humildemente me educaram. Um muito obrigado por isso e por todo o apoio ao longo da vida. Aos meus irmãos, os eternos rebeldes, obrigado por me fazerem ter alguém com quem me preocupar.

De seguida, quero agradecer à minha namorada, Marta. Servidores, processos e sistemas distribuídos já não são termos incompreensíveis, pois consegue passar horas a ouvir-me falar disso. Partilhei todos os bons e maus momentos pelos quais passei e nunca deixei de ver o seu sorriso. Obrigado por seres para mim.

Todos os meus amigos sabem o quanto me são importantes, mas não posso deixar de agradecer tudo o que fizeram, e ainda fazem, por mim. Por tempos em que amigos de verdade são cada vez mais um sinal de riqueza, considero-me um multimilionário. Um muito obrigado a todo o "Peeps" e em especial ao Pimba, à Soraya, ao Bisgas, ao Calvin, ao Passos e ao Oliveira, porque sei que davam só o que fosse preciso por mim, sem hesitar, e porque sabem que eu faria o mesmo, sem sequer pestanejar. Um abraço.

Ao meu orientador, o Professor Alysson Bessani, um muito obrigado por toda a ajuda e orientação que me deu. Mais do que essa ajuda, tenho em consideração a forma aberta e comunicativa com que trocávamos ideias, nenhuma delas sendo descabida o suficiente para ser motivo de reprimenda ou crítica negativa.

Quero, também, deixar uma palavra de agradecimento ao Professor José Orlando Pereira, da Universidade do Minho, pela especificação da camada de coordenação da arquitectura *ReD*. Esta camada foi especialmente importante para este projecto, definindo o conteúdo da secção 3.2.

Ao Pintinho e ao Maiur, que quando os tempos ficaram mais curtos e complicados, não se esqueceram de serem os bons amigos e bons colegas que sempre foram. Um enorme obrigado.

Obrigado, também, ao colega João Sousa pelo apoio dado em relação ao SMaRt.

Durante o meu tempo na FCUL, fiz muitas amizades, mas também alguns inimigos. Passei por muitas peripécias, dificuldades, adversidades e desilusões, mas também muitas alegrias, felicidades e sucessos. Foi o melhor tempo da minha vida. Nunca me diverti

tanto e nunca aprendi tanto sobre a vida, como nestes últimos anos. Obrigado a todos os que tive o prazer de conhecer, amigos ou inimigos.

Por fim, à PRAXE por me ter ensinado que somos caloiros da vida para sempre e por ter sido uma importante parte da minha vida académica, proporcionando-me momentos mesmo muito bons, deixo o meu respeito.

Para os melhores tempos e pessoas.

Resumo

Existem duas arquitecturas básicas para a replicação de bases de dados. Na arquitectura chamada *shared-storage* as réplicas partilham um sistema de armazenamento, pelo que é necessário algum tipo de coordenação entre as réplicas para o acesso aos dados armazenados. Na outra arquitectura, chamada *shared-nothing*, cada réplica acede apenas ao seu armazenamento particular (que até pode ser um disco local), pelo que a necessidade de coordenação é menor.

O projecto *ReD* (*Resilient Databases*) propõe uma nova arquitectura híbrida para as bases de dados replicadas. Todas as réplicas têm uma cópia local da base de dados e apenas uma escreve no sistema de armazenamento partilhado, ou seja, apenas uma réplica é considerada escritora. Para conseguir tal facto, mesmo na presença de falhas e de novas réplicas, esta arquitectura contém uma camada de coordenação.

Neste projecto fazemos uma análise de vários serviços de coordenação existentes e procuramos uma comparação entre eles para uma melhor compreensão das opções disponíveis, com o objectivo de escolher qual o mais indicado para ser usado na implementação da camada de coordenação da arquitectura do *ReD*.

Após esse estudo, desenvolvemos e concretizámos um algoritmo de eleição de líder que utiliza o serviço de coordenação *ZooKeeper* para a implementação dessa camada de coordenação.

Na segunda fase do projecto, apresentamos uma proposta tolerante a faltas bizantinas para essa mesma camada. Neste caso, foi também desenvolvido e concretizado um algoritmo de eleição de líder, mas utilizando o serviço de coordenação *DepSpace*. No entanto, devido a algumas limitações deste serviço, foi necessário o desenvolvimento de uma nova versão com funcionalidades que permitissem essa concretização.

Com o estudo dos serviços de coordenação e com as implementações destes algoritmos, mostramos que é possível reduzir o esforço no desenvolvimento de sistemas distribuídos complexos, pois os programadores podem concentrar-se no desenvolvimento do serviço em si.

Palavras-chave: serviços de coordenação, tolerância a faltas, zookeeper, depspace, eleição de líder

Abstract

There are two basic database replication architectures. Sharing a storage system between servers is called shared-storage architecture and it needs some sort of coordination so servers can access the storage. With the shared-nothing architecture, servers can only access their private storage (like a local hard disk), which means that it needs less coordination between them.

ReD (Resilient Databases) project proposes a new hybrid database replication architecture. In this new architecture all servers have a local database copy, but only one writes in the shared storage and this is accomplished with a coordination layer.

In this project we analyse and compare some existing coordination services in order to understand which is best to use in that coordination layer implementation.

After the study, we implemented the layer by developing a leader election algorithm using ZooKeeper as the coordination service.

In the second phase of this project we propose a byzantine fault tolerant approach for the same layer. Following first phase procedure, a leader election algorithm was developed and implemented, but in this case using DepSpace as the coordination service. However, due to some limitations in this service, we developed a new version with features that allowed this implementation.

The study of coordination services and implementations of these algorithms, show that it is possible to reduce effort on the development of complex distributed systems, because programmers can concentrate on the development of the service itself.

Keywords: coordination services, fault tolerance, zookeeper, depspace, leader election

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Os Sistemas de Gestão de Bases de Dados (SGBD) são uma força confiável na indústria das tecnologias de informação. Estes sistemas são um componente fundamental na escalabilidade e disponibilidade das aplicações. Os fundamentos dos SGBD foram definidos nos anos 70, pelo que a sua adaptação face aos desafios das aplicações modernas é uma importante área de investigação. Assim, um dos desafios diz respeito à escalabilidade e fiabilidade dos SGBD. Muitas aplicações armazenam grandes volumes de dados nestes sistemas ao mesmo tempo que requerem uma alta disponibilidade destes mesmos dados. A replicação da base de dados num conjunto de servidores é essencial para garantir essa disponibilidade e a divisão do esforço de armazenamento e processamento.

Efectivamente, existem duas arquitecturas básicas para a replicação de base de dados.

Na primeira, designada por *shared-storage* [28], os diversos servidores usam um sistema de armazenamento partilhado, por exemplo, uma rede de discos (como podemos ver na figura 1.1). Neste modelo, o número de servidores depende apenas da carga de trabalho e da disponibilidade que se pretende assegurar. O armazenamento é configurado de acordo com a capacidade dos discos e a quantidade de dados a armazenar. No entanto, uma aproximação deste tipo levanta inúmeros problemas devido à necessidade de coordenação entre os servidores, para manter a consistência dos dados partilhados, e requer complexos protocolos de exclusão mútua distribuída tolerante a faltas com altos custos de desenvolvimento.

Por outro lado, na arquitectura designada por *shared-nothing* [45] [44], cada servidor acede ao seu sistema de armazenamento particular, como podemos ver na figura 1.2, normalmente o seu disco rígido local. Estes sistemas baseiam-se em replicação consistente [38] onde é utilizado um protocolo de coordenação para distribuir as actualizações de forma coerente por todos os servidores, como por exemplo a ordenação total de escritas [25]. O desempenho e disponibilidade resultantes são bastante bons, especialmente com cargas de trabalho onde predominam leituras. No entanto, apesar de ser mais simples que o modelo de partilha de discos, esta arquitectura necessita de uma cópia física dos dados em cada servidor. Em sistemas de grande escala, tal facto impõe um custo de *hardware* e

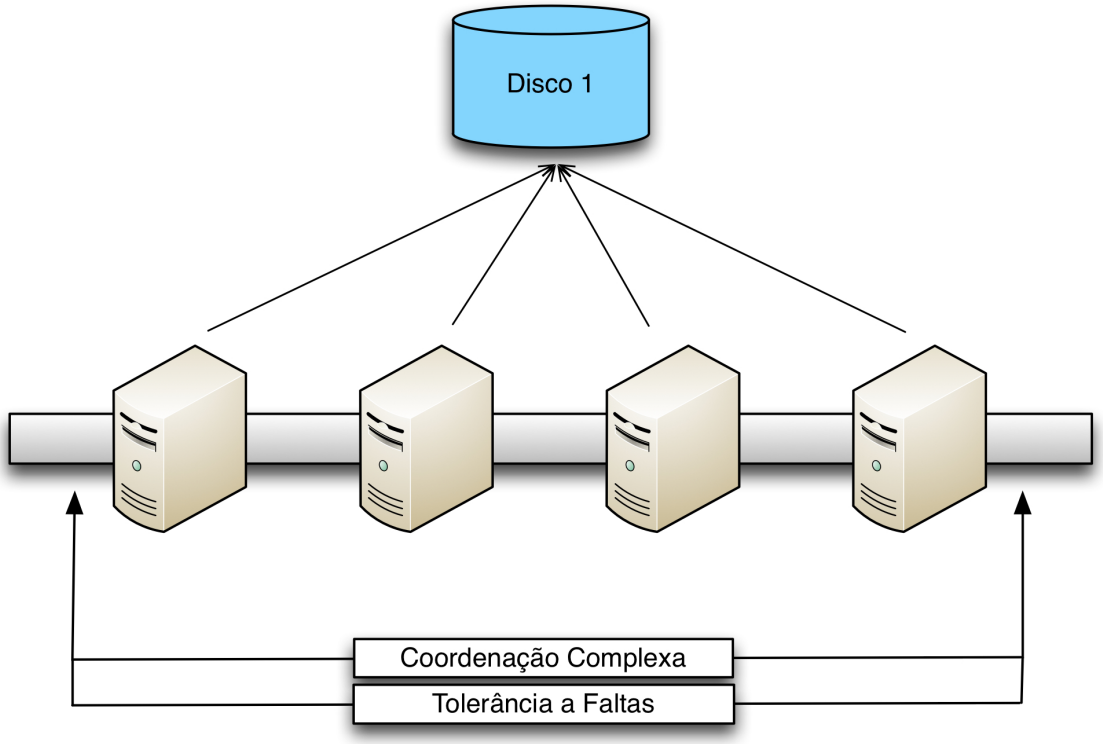

Figura 1.1: Arquitectura *Shared-Storage*

de operação que ultrapassa a sua vantagem inicial.

O projecto *ReD* (*Resilient Databases*) [1] é uma colaboração entre a Universidade do Minho e a Faculdade de Ciências da Universidade de Lisboa (FCUL) [1], que propõe uma arquitectura híbrida (*shared-storage* e *shared-nothing*) para a replicação de SGBD. Esta aproximação consiste em combinar o protocolo de replicação com um sistema de gestão de volumes *copy-on-write* [46]. Nesta nova abordagem, cada servidor tem o seu próprio disco para escrita temporária de actualizações. Um destes servidores, considerado líder, é o único que escreve no sistema de armazenamento partilhado. Combinamos, então, os dois modelos previamente mencionados: sistema de armazenamento proporcional ao tamanho da base de dados (dados armazenados num sistema partilhado, os discos locais são apenas para *cache*) e simplicidade (não há exclusão mútua distribuída). Esta arquitectura, utilizada no projecto *ReD*, é analisada em pormenor na secção 3.1.

Este Projecto em Engenharia Informática (PEI) visa concretizar a camada de coordenação entre as réplicas da arquitectura *ReD*, que garante a capacidade do sistema operar de forma consistente mesmo na presença de falhas. Um ponto distinto deste projecto é a utilização de serviços de coordenação na concretização desta camada [4] [8] [14] [29].

O uso destes serviços facilita a sincronização distribuída de processos.

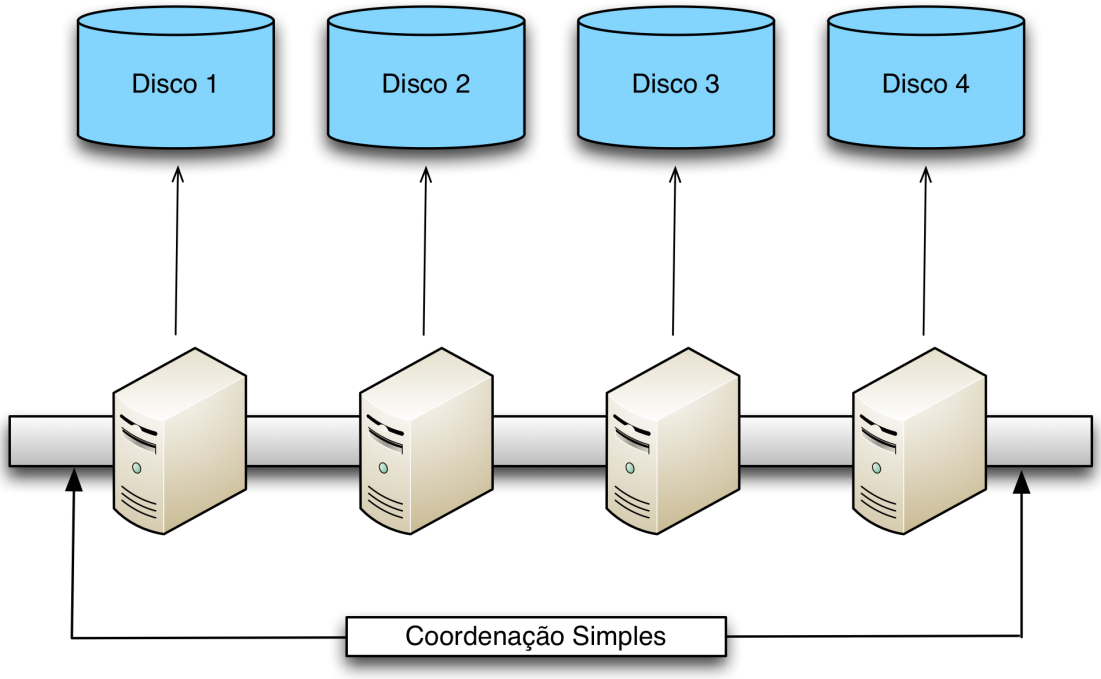

Figura 1.2: Arquitectura *Shared-Nothing*

1.1 Motivação

Como vimos na secção anterior, na nova arquitectura híbrida *ReD* apenas uma réplica escreve no armazenamento partilhado. Esta réplica considerada escritora pode falhar, logo é necessária uma camada de coordenação que, entre outras coisas, elege uma réplica escritora e que detecta faltas.

Nos últimos anos, os serviços de coordenação foram ganhando visibilidade na ajuda ao desenvolvimento de sistemas distribuídos complexos e podem ser usados, por exemplo, para a concretização da coordenação das réplicas no *ReD*.

Os modelos de sistema em que a coordenação nessa arquitectura pode ser concretizada constituem um desafio interessante, visto que é necessário a utilização de um serviço de coordenação para o *crash-stop* e outro para um modelo tolerante a faltas bizantinas.

1.2 Objectivos

O projecto *ReD* tem como objectivo o desenvolvimento de uma arquitectura híbrida (*shared-storage* e *shared-nothing*), genérica, robusta e pouco dispendiosa, realizada em parceria com a Universidade do Minho.

O objectivo primordial do nosso projecto é o desenho e a concretização da camada de coordenação das réplicas na arquitectura *ReD*. Será usado um serviço de coordenação

como base dessa camada.

Os objectivos específicos deste PEI são:

- Estudar os principais serviços de coordenação existentes e suas características;
- Concretizar a camada de coordenação utilizada na arquitectura *ReD*, para o modelo *crash-stop*;
- Propor uma camada de coordenação similiar à anterior, porém tolerante a faltas bizantinas.

Esta arquitectura híbrida é concretizada contemplando faltas por paragem (modelo *crash-stop*), através da utilização do serviço de coordenação *ZooKeeper*, e faltas bizantinas, através da utilização do serviço de coordenação *DepSpace*. No entanto, é necessário o desenvolvimento de uma nova versão deste serviço, concretizada sobre a biblioteca de replicação *BFT-SMaRt*, por forma a tolerar, de facto, faltas bizantinas.

1.3 Estrutura do documento

Este documento está organizado da seguinte forma:

- Capítulo 2 - Introdução aos serviços de coordenação, descrição de alguns serviços de coordenação existentes, bem como a sua comparação;
- Capítulo 3 - Explicação do projecto *ReD* e arquitectura utilizada e apresentação de uma solução de coordenação e sua prova informal;
- Capítulo 4 - Apresentação de uma solução de coordenação tolerante a faltas bizantinas e melhorias ao serviço de coordenação *DepSpace*;
- Capítulo 5 - Conclusão e trabalhos futuros.

Capítulo 2

Serviços de Coordenação

Nos sistemas distribuídos modernos a partilha de dados é um requisito comum. Esta partilha é alcançável através de um protocolo que normalmente assenta sobre múltiplas trocas de mensagens. No entanto, este tipo de aproximação leva a um modelo de programação propenso a muitos erros e frustração. Os serviços de coordenação vêm colmatar este modelo de programação inadequado para sistemas distribuídos.

2.1 Introdução aos Serviços de Coordenação

Os processos podem comunicar de várias formas diferentes, entre as quais, através de troca explícita de mensagens, acesso a memória partilhada ou até recursos partilhados. Estes são os grandes paradigmas que suportam a comunicação e a sincronização. Como consequência, os processos devem usar esses paradigmas para se coordenarem, garantindo, por exemplo, que o acesso aos recursos partilhados é funcional (exclusão mútua [48]).

Um serviço de coordenação é um serviço para manter informações de controlo e configuração e realizar sincronização distribuída, sendo portanto útil de uma forma ou de outra em aplicações distribuídas.

Os serviços de coordenação apresentam uma forma de retirar ao programador a responsabilidade de coordenar os processos, dando espaço para que este se foque no desenvolvimento da aplicação propriamente dita, reduzindo consideravelmente o esforço e as linhas de código necessárias.

Como se pode verificar na figura 2.1, um serviço de coordenação é diferente de um serviço de comunicação em grupo (como, por exemplo, o *Appia* [36]). Um serviço de comunicação em grupo consiste numa abstracção em que um participante envia uma mensagem e todos os membros do grupo a recebem (são suportadas diferentes qualidades de serviço) [32].

Um serviço de coordenação oferece mais do que apenas o envio de mensagens entre todos os participantes. De facto, oferece abstracções fortes o suficiente para que proces-

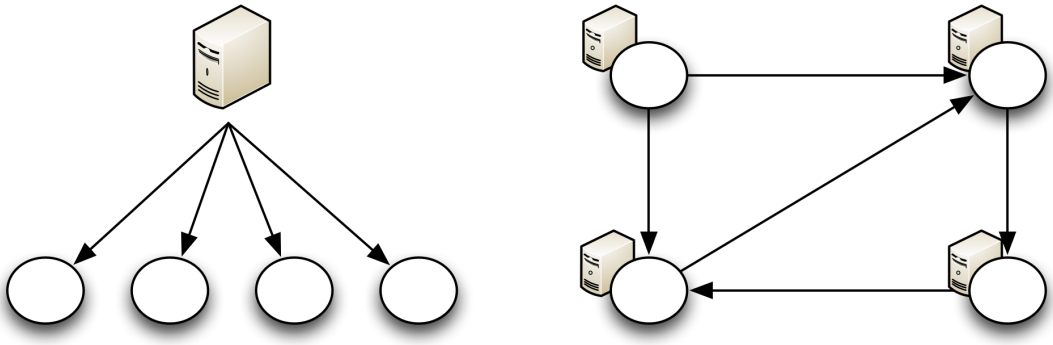

Figura 2.1: Serviço de coordenação e serviço de somunicação em grupo

sos possam, de forma determinista, partilhar estado, eleger um líder, verificar quantos processos estão ligados ao serviço e até concretizar comunicação em grupo.

Existem, como foi já mencionado, vários tipos de serviços de coordenação. Efectivamente, existem serviços de baseados em espaço de tuplos (*DepSpace* [8], *JavaSpaces* [20], *IBM TSpaces* [31]), em filas (*Amazon Simple Queueing Server* [5]) em sistemas de ficheiros com *locks* (*Chubby* [14]) e também em espaços de nomes hierárquicos (*ZooKeeper* [29]).

2.1.1 Porquê usar?

Uma primeira aproximação à coordenação de processos é a configuração de sistemas distribuídos. A configuração consiste num conjunto de parâmetros, normalmente uma lista num ficheiro, que os processos seguem. No entanto, esta aproximação não é suficiente em ambientes em que as máquinas estão sempre a mudar dinamicamente. Podemos, então, distribuir esse ficheiro de configuração por forma a que os processos verifiquem, periodicamente, se houve alterações. No entanto, torna-se muito complicado a gestão de uma coordenação deste tipo, uma vez que podem existir diferentes ficheiros de configuração ou subsistemas que usem ficheiros diferentes. Neste caso, pode ser necessário saber quais as configurações usadas pelos processos e voltar atrás caso alterações de configuração não sejam aceites correctamente.

Também se poderia adoptar outra aproximação em que cada processo teria um servidor *web* embutido, para que fosse possível observar certas métricas e efectuar alterações, em tempo-real, em cada processo. Seria uma aproximação correcta e suficiente para um simples processo, mas muito difícil e trabalhosa para manipular um grande conjunto de processos.

Os serviços de coordenação podem manter toda a configuração necessária para ser acedida pelos processos do sistema.

Outro problema mais relevante e complexo que os serviços de coordenação conseguem resolver é o consenso [48]. Esta solução permite a implementação de funcionalidades importantes nos sistemas distribuídos, em áreas como a coordenação de réplicas. Nomeadamente, com um serviço que ofereça consenso, é possível a implementação de soluções de coordenação como difusão de mensagens em ordem total [25] e eleição de líder.

A redução de complexidade de criação de aplicações em sistemas distribuídos através do uso de serviços de coordenação liberta os programadores por forma a focar o esforço no desenvolvimento da aplicação.

É, ainda, possível que os serviços de coordenação utilizados ofereçam abstracções já seguras e fidedignas. Como se vai ver, o *DepSpace* (secção 2.3.4), oferece um espaço de tuplos seguro, que serve de base e facilita a construção de sistemas distribuídos confiáveis, já que os programadores começam o desenvolvimento sobre algo em que podem confiar. Deste modo, a utilização de serviços de coordenação é, então, uma mais valia para os programadores.

2.1.2 Exemplos de uso

Quando se desenvolve sistemas distribuídos pode existir a necessidade de, por exemplo, partilha de estado entre máquinas, tolerância e detecção de faltas ou eleição de um líder. Estas propriedades são relativamente fáceis de concretizar num serviço pequeno, mas não em serviços de larga escala, sendo que o uso de serviços de coordenação facilita a sua implementação.

A primeira e segunda fases deste projecto são também exemplos do uso dos serviços de coordenação, pois permitiram-nos a implementação de um protocolo de eleição de líder com requisitos acrescidos, através de algoritmos de não mais que uma página.

Seguidamente iremos analisar alguns dos serviços de coordenação (secção 2.3), já referidos anteriormente, mostrando assim a utilidade de cada um baseada nas suas características.

2.2 Abstracções e Primitivas para Serviços de Coordenação

Os serviços de coordenação usam abstracções e primitivas que lhes permitem fornecer um modelo de programação adequado ao desenvolvimento de sistemas distribuídos. O entendimento dessas abstracções e primitivas leva a uma maior facilidade na escolha e uso do serviço mais adequado às necessidades do sistema a desenvolver.

2.2.1 Abstracções da Memória

Por abstracções denotamos a organização da memória partilhada oferecida pelo serviço de coordenação.

Registos partilhados

São endereços de memória onde se pode armazenar dados, sem que seja imposta uma estrutura. Este tipo de abstracção permite que os dados armazenados sejam tratados de forma relativamente independente entre endereços de memória [4].

Espaço de nomes hierárquico

Pode ser instanciado como um sistema de ficheiros restrito (suporte apenas para ficheiros pequenos e operações básicas) ou como um serviço de nomes (onde se pode associar dados a nomes).

Esta é uma forma simples de adquirir coordenação entre processos, visto que a criação e manipulação de ficheiros ou de nomes é um modelo bastante familiar para a maioria dos programadores.

Espaço de tuplos

Um espaço de tuplos é uma abstracção de memória partilhada sobre um sistema distribuído [21]. Um tuplo é um conjunto ordenado de campos e são inseridos, lidos e removidos do espaço. Não existe estrutura e os tuplos são acedidos por conteúdo.

Esta ideia proporciona um modelo de programação simples e tem sido usado na construção de sistemas distribuídos complexos.

2.2.2 Primitivas de Sincronização

As primitivas de sincronização definem o poder do serviço de coordenação para resolver problemas em sistemas distribuídos.

Minitransacções

Uma minitransacção é uma primitiva que permite, de forma atómica, aceder e alterar condicionalmente dados num endereço de memória [4]. São úteis para melhorar o desempenho do sistema, porque permitem criar conjuntos de actualizações (reduzindo o número de mensagens na rede) e podem ser executadas no protocolo *commit* das transacções comuns em base de dados (inicia, executa e faz *commit* em apenas duas mensagens).

É uma primitiva de sincronização poderosa pois permite operações de leitura, escrita e comparação, todas com a propriedade de serem atómicas. Torna também possível a resolução de consenso.

Locks e *leases*

Um *lock* [18] é um mecanismo de sincronização, que oferece controlo de concorrência, para acesso a um recurso. Esta primitiva pode ser utilizada por praticamente todas as abstracções, por exemplo, num serviço de coordenação que use uma abstracção sistema de ficheiros (espaço de nomes hierárquico) um ficheiro criado pode ser um *lock*.

Um *lease* [23] é um contracto que especifica direitos sobre um recurso, durante um intervalo de tempo. O uso de *leases* garante que quem possui um *lease* válido, tem o controlo sobre o recurso protegido. Este mecanismo de sincronização é semelhante a um *lock*, com a diferença que os *locks* não estão limitados temporalmente, ou seja, outro processo só pode aceder aos recursos quando o processo que fez *lock* o libertar. Posto isto, um serviço de coordenação que use este mecanismo é responsável por toda a gestão das *leases*. Esta gestão dá um enorme poder de sincronização uma vez que o serviço de coordenação pode atribuir e retirar *leases*, sendo que, deste modo, cabe ao programador utilizar essa gestão da forma que melhor se adequar ao serviço que está a desenvolver.

Espaço de tuplos estendido

Conforme já mencionado, as operações suportadas por um espaço de tuplos são normalmente inserir, ler ou remover um tuplo do espaço, ou seja, *OUT*, *RDP* e *INP*, respectivamente. No entanto, uma abstracção espaço de tuplos pode suportar uma outra operação chamada *CAS* (*condicional atomic swap*) [8]. É uma operação atómica que permite procurar se já existe um dado tuplo e, em caso negativo, inserir outro tuplo. Esta operação é importante pois consiste numa primitiva de coordenação que permite resolver o problema do consenso. Para tal, apenas é necessário que cada processo tenha um valor para propor, podendo comparar, numa só instrução, se já existe algum valor proposto e caso isso não se verifique, impor o seu.

A existência desta primitiva faz com que um espaço de tuplos seja capaz de resolver consenso.

Criação ordenada de objectos

Esta primitiva baseia-se num contador atómico. Se muitos processos criarem objectos ao mesmo tempo, esta primitiva garante que estes objectos ficam sempre ordenados com um número de sequência.

Utilizando, por exemplo, a abstracção de um sistema de ficheiros e esta primitiva, é possível resolver o problema do consenso, bem como a implementação de outras formas de sincronização. Se todos os processos criarem um ficheiro, o serviço de coordenação trata de acrescentar um número de sequência e garante que nunca há dois ficheiros com o mesmo número de sequência. Desta forma, pode ser definido, por exemplo, que o valor decidido por todos os processos é sempre o valor de sequência mais baixo que exista.

2.3 Análise de serviços de coordenação

Nesta secção apresentamos uma análise de alguns dos serviços de coordenação que se consideraram adequados e importantes para este estudo, com o objectivo de oferecer um panorama geral do estado dos serviços de coordenação existentes.

2.3.1 Sinfonia

O *Sinfonia* [4] é um serviço que permite partilhar dados de aplicações de uma forma tolerante a faltas, escalável e consistente.

A abstracção utilizada por este serviço de coordenação são os registos partilhados (explicado na secção 2.2.1). No caso do *Sinfonia*, estes registos consistem num conjunto de nós de memória (*RAM* ou disco, dependendo da necessidade da aplicação) com um espaço de endereçamento linear. Os dados são referenciados por um par *id* do nó e endereço e a sua manipulação é feita através de um mecanismo básico de escrita e leitura de *bytes*. Não oferece suporte para manipulação de múltiplos nós em simultâneo.

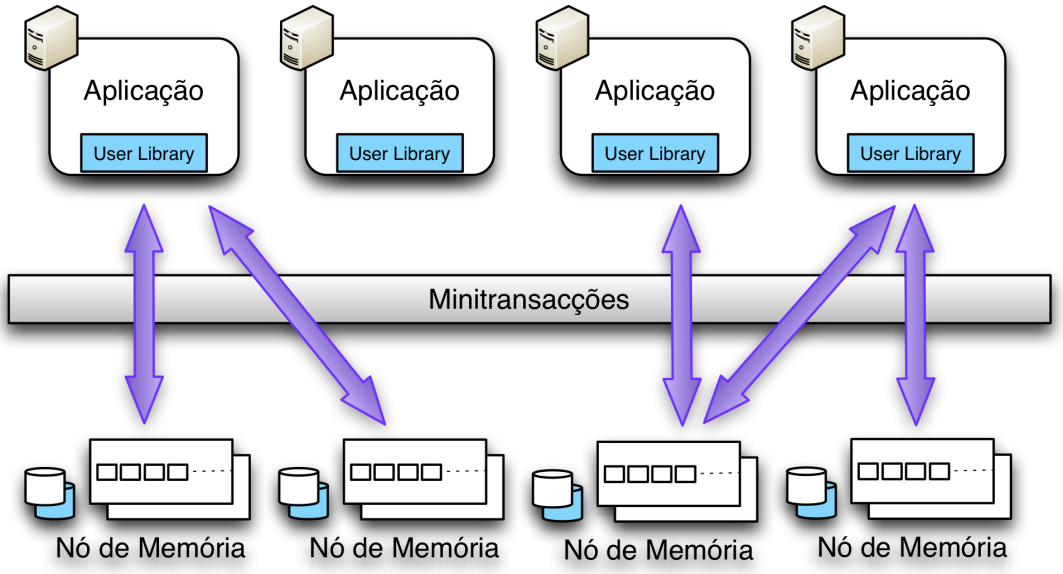

Figura 2.2: *Sinfonia*

A primitiva de coordenação utilizada por este serviço são as minitransacções. Estas, constituem outro mecanismo de manipulação de nós, como podemos verificar na figura 2.2, e permitem actualizações *ACID* (*Atomicity, Consistency, Isolation, Durability*) [26]. Uma minitransacção executa apenas a parte *commit* do protocolo *two-phase commit* [37].

Em termos de persistência, os nó podem ser guardados num disco. Para além disso, é feito um *backup* dos dados dos nós. Este *backup* obedece a um conjunto de passos:

1. Cada nó mantém um *log* que guarda actualizações recebidas (se o nó for replicado, só guardam no *log* se todos os participantes votarem para fazer *commit*);

2. Cada nó de memória adquire um *lock* bloqueante, por ordem de *id* (para evitar *deadlocks*);

3. Todos os nós de memória actualizam o seu estado até à última transacção presente no respectivo *log* (é possível continuar a receber transacções porque podem ser escritas no *log* de actualizações);

4. Assim que todos os nós tenham atingido o estado da última transacção presente no *log*, o disco pode ser copiado ou é gerado e guardado um *snapshot*.

O *Sinfonia* não faz nenhum tipo de *cache*. De qualquer forma, é possível criar um sistema deste tipo utilizando as minitransacções, pois fazem validação e aplicação de alterações de uma forma atómica.

Um nó de memória pode ser replicado para efeitos de disponibilidade e de tolerância a faltas. Essa replicação é *primary-copy* [39], ou seja, uma réplica recebe as actualizações vindas de uma réplica primária e actualiza o seu próprio *log*. Este tipo de replicação baseia-se na sincronia para perceber que uma réplica primária falhou, pelo que, usado em sistemas assíncronos pode levar a falsas notificações de falhas. Isto faz com que duas réplicas se consideram primárias ao mesmo tempo, perdendo-se a consistência do sistema. Para resolver este problema, o *Sinfonia* usa *lights-out management* (ligar ou desligar uma máquina remotamente) para desligar a réplica primária que provocou o *fail-over*, ficando assim apenas uma réplica que se considere primária.

Este serviço pode executar milhares de minitransacções por segundo com uma baixa latência, usando apenas um nó, e escala bem até centenas de nós. Entre as aplicações concretizadas com este serviço de coordenação podemos citar um sistema de ficheiros distribuídos e um sistema de comunicação em grupo [4].

2.3.2 Chubby

O *Chubby* é um serviço de coordenação, desenvolvido pelo *Google*, com ênfase na disponibilidade e confiabilidade. Usa uma abstracção espaço de nomes hierárquica, nomeadamente um sistema de ficheiros. Esta abstracção fornece ao cliente uma interface que permite operações básicas de escritas e leituras de ficheiros, mas melhorada com *locks* e notificações de eventos. Não se pode mover ficheiros nem há datas de modificação de directórios. Existem, sim, nós permanentes ou efémeros, podendo qualquer um destes, ser um *lock* de escrita ou de leitura. Em termos de segurança, o sistema de ficheiros tem uma *Access Control List* (*ACL*), com permissões de escrita, leitura e alterações a essa lista.

A primitiva de coordenação que este serviço oferece aos programadores são os *locks*. Estes, permitem que os clientes se sincronizem e concordem sobre informação básica

dos seus ambientes. No entanto, o custo de transferência de um *lock* entre clientes é elevado e podem ser necessários procedimentos de recuperação, logo, é favorável que um *lock* sobreviva a uma falha de servidor. Para colmatar este problema, o *Chubby* oferece *coarsed-grained locks* [47]. Os programadores têm a responsabilidade de ajustar o tempo de *lease* dos *locks* para que o serviço suporte a carga pretendida.

Existem *locks* exclusivos (escrita) e *locks* partilhados (leitura). As operações que precisem de *locks* têm um número de sequência, passado ao servidor *Chubby* para que este possa verificar se os *locks* são ou não válidos. No entanto, todo este processo é lento, pelo que é necessário uma boa gestão dos *locks*. Este serviço usa o conceito de *lock-delay*, garantindo que se um *lock* ficar livre por consequência de uma falha de quem o tinha, o servidor não permite que outros o adquirem imediatamente. Durante um tempo arbitrário, o servidor mantém a esperança que o processo cliente que tinha o *lock* recupere, para que o possa manter.

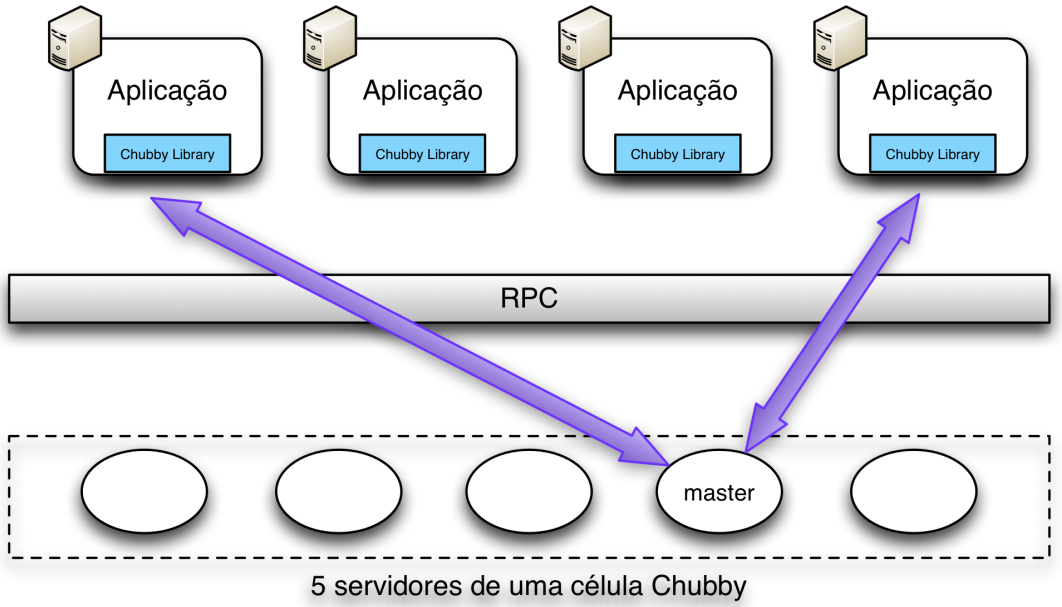

Figura 2.3: Arquitectura *Chubby*

Como podemos ver na figura 2.3, o sistema está estruturado em servidores *Chubby*, bibliotecas *Chubby* (no cliente) e a comunicação é feita através de chamadas a procedimentos remotos. O sistema é replicado, sendo que normalmente 5 réplicas constituem uma célula *Chubby* (tolerante a falhas até duas delas). Uma destas réplicas é o líder, durante um *master lease* (eleito através de um consenso distribuído). Apenas o líder lê e escreve na base de dados, pelo que as restantes se limitam a copiar as actualizações que recebem. No entanto, as escritas são propagadas para todas as réplicas e aceites quando confirmadas por uma maioria.

Uma sessão entre uma célula *Chubby* e um cliente *Chubby* existe durante um intervalo de tempo, sendo mantida por mensagens *KeepAlive* periódicas. Tanto o cliente, como o servidor têm um *lease* de sessão. Quando um líder falha, outro é eleito e durante essa eleição as *leases* dos clientes podem expirar, começando um *grace period* que, se também expirar, o cliente abandona a sessão.

Em termos de persistência, além de manter os dados guardados em ficheiros locais, são feitos *backups* para um *Google File System* [22], em intervalos de poucas horas. É permitido fazer *mirroring* (cópia ficheiros de uma célula para outra), uma vez que é relativamente rápido, devido aos ficheiros serem pequenos.

Se uma réplica não recuperar em pouco tempo, é seleccionada outra máquina através de um sistema simples de substituição, actualizando as tabelas *DNS* e o endereço *IP*. Esta nova réplica recebe uma cópia recente da base de dados, através de um *backup* guardado em servidores de ficheiros, e as actualizações que ainda não se encontram no *backup*, enviadas por réplicas activas.

O *Chubby* permite que os clientes guardem ficheiros pequenos, para serem usados mais tarde, ou seja, pode ser feita uma *cache* do lado do cliente. Para manter a consistência, a modificação de um ficheiro é bloqueada enquanto o líder envia invalidações a todos os clientes que o tinham guardado. O sistema permite também fazer *cache* de *locks*, ou seja, um cliente pode ter um *lock* mais tempo do que o necessário, na esperança que venha ainda a precisar dele.

Este serviço oferece também um mecanismo de notificação, baseado em eventos. Estes, são entregues de forma assíncrona ao cliente, após a acção correspondente ter acontecido. Por exemplo, quando um cliente pede um *lock*, é enviado um evento a notificar o processo que o tem adquirido. Deste modo, um *lock* pode apenas ser libertado quando for necessário a algum cliente.

Por fim, o *Chubby* é um serviço de coordenação muito usado como repositório para informações de configuração, usa *cache* consistente no cliente (o que reduz carga no servidor), notificações de actualizações e uma interface familiar de um sistema de ficheiros.

2.3.3 Zookeeper

O *ZooKeeper* [19] [29] é um serviço de coordenação de processos de aplicações distribuídas simples e com alto desempenho. Permite que processos distribuídos se coordenem através de um espaço de nomes hierárquico, sendo esta a abstracção utilizada. Mais concretamente, consiste num conjunto de nós de dados, semelhante a um sistema de ficheiros em árvore. Os nós, chamados *znodes*, são também mantidos em memória, permitindo apresentar um desempenho excelente em operações de leitura.

A primitiva de coordenação oferecida neste caso é a criação ordenada de objectos (ver secção 2.2.2). Neste caso, os objectos são *znodes* regulares ou efémeros, manipulados através de uma API.

Este serviço de coordenação pode ser replicado em várias máquinas, evitando ser um ponto único de falha e estrangulamento do sistema, garantindo, deste modo, uma alta disponibilidade. As réplicas conhecem-se umas às outras e mantêm uma imagem do estado do sistema e um registo (*log*) de transacções. Uma das réplicas é o líder, que coordena as escritas, e todas as outras podem tratar de leituras. Assume-se falhas por paragem e que as réplicas podem recuperar. O serviço *ZooKeeper* está disponível enquanto uma maioria de servidores estiverem disponíveis e, caso uma ligação do cliente a um servidor seja quebrada, este liga-se a um outro servidor.

É garantida linearidade das escritas (respeito de precedências) usando o *Zab* [41], um protocolo de difusão com ordem total [25]. A linearidade oferecida pelo *ZooKeeper* é diferente do conceito original de linearidade, já que, por definição, linearidade permite que um cliente realize uma operação de cada vez (um cliente é uma *thread*). No entanto, o *ZooKeeper* oferece linearidade assíncrona, a qual permite que um cliente invoque várias operações ao mesmo tempo, ordenadas em *FIFO*.

Para efeitos de persistência, forçam-se as actualizações (escritas) em disco, antes de serem aplicadas em memória. Cada réplica contém uma cópia, em memória, do estado do *ZooKeeper* (uma base de dados local em memória que contém toda a árvore de dados). Quando um servidor recupera de uma falha, recebe um *snapshot* do estado e todas as mensagens recebidas após esse *snapshot*, evitando assim, ter de receber as mensagens desde o início. Estes *snapshots*, enviados às réplicas quando recuperam, são feitos periodicamente e constituem um *backup* que pode ou não corresponder a um estado actualizado do *ZooKeeper*. Por esta razão, são chamados *fuzzy snapshots*. No entanto, não há problema se corresponderem a um estado antigo, já que uma réplica que recupere recebe também todas as mensagens processadas depois desse mesmo *snapshot*.

Uma alteração persiste, perante qualquer número de falhas, desde que um quorum de servidores consiga recuperar.

O *ZooKeeper* oferece um sistema de notificação baseado no conceito de *Watch*. Este mecanismo, pode ser usado, por exemplo, para efeitos de *cache* ou detecção de falhas, dependendo das necessidades da aplicação a desenvolver. Ao colocar um *Watch* num objecto, é activado um evento de notificação quando acontecer uma alteração. Existem, no entanto, algumas nuances, que não podemos esquecer ao usar este sistema.

- Cada *Watch* colocado é activado apenas uma vez. Ou seja, após a recepção de uma notificação se quisermos obter uma nova notificação, é necessário colocar outro *Watch* no objecto;

- Os clientes podem receber as notificações em tempos diferentes (devido a assincronia), mas recebem-nas pela mesma ordem;

- Um nó pode mudar, várias vezes, durante o tempo do cliente receber a notificação de alteração e colocar outro *Watch*, ou seja, é possível perder alterações a um nó.

Em termos de detecção de falhas, o cliente envia mensagens, chamadas *heartbeats*, para manter a ligação aberta. Para detectar se o cliente falhou, o servidor usa um *timeout*. Se forem enviados pedidos frequentemente não se enviam *heartbeats*. Assim como no *Chubby*, é necessário uma maioria de servidores para manter o serviço disponível, i. e., $n \geq 2f + 1$, em que *f* é o número de faltas que *n* servidores conseguem tolerar.

Como as leituras são processadas localmente em cada réplica, escalam linearmente à medida que vão sendo adicionados mais servidores.

2.3.4 DepSpace

O *DepSpace* [8] é um serviço de coordenação que utiliza uma abstracção de espaço de tuplos. Esta, foi escolhida pela sua simplicidade, endereçamento por conteúdo e comunicação desacoplada. Oferece operações básicas para a manipulação do espaço, ou seja, inserir, ler e remover tuplos (sequências finitas de valores) e ainda suporte para múltiplos espaços de tuplos.

A primitiva de coordenação que este serviço oferece é uma extensão ao espaço de tuplos representada pela operação *CAS*, que é suficientemente poderosa para resolver o problema do consenso, como foi visto anteriormente na secção 2.2.2.

O modelo de sistema assenta sobre comunicação cliente/servidor, através de canais ponto-a-ponto autenticados e confiáveis (*TCP* e *MACs*[1]), assumindo que a rede pode perder, corromper e atrasar mensagens, mas que não pode interromper a comunicação entre processos correctos indefinidamente. É necessário um sistema parcialmente sincrono, para garantir *liveness*, visto que usa um protocolo de consenso baseado em *Paxos* bizantino [34].

Este serviço consiste numa série de camadas, como podemos ver na figura 2.4. De seguida, explicamos as mais importantes.

- Replicação - Baseia-se no modelo de replicação de máquinas de estado [42]. Todas as réplicas têm o espaço de tuplos guardado em memória e executam a mesma sequência de operações de uma forma determinista. São necessárias $3f+1$ réplicas, em que *f* é o número de faltas bizantinas que o sistema consegue tolerar. O cliente envia um pedido através de um algoritmo de difusão com ordem total tolerante a faltas bizantinas [15] e espera *f+1* respostas com o mesmo conteúdo, cada uma vinda de uma réplica diferente;

- Confidencialidade - A replicação é vista como um impedimento à confidencialidade (se a informação secreta existe em mais réplicas, há uma maior probabilidade de deixar de ser confidencial). O *DepSpace* usa um esquema *Publicly Verifiable Secret Sharing* (*PVSS*) [43], em que cada servidor tem uma parte do segredo, neste caso um tuplo, e são necessárias no mínimo *f+1* partes para recuperar o tuplo original;

[1] Message Authentication Codes

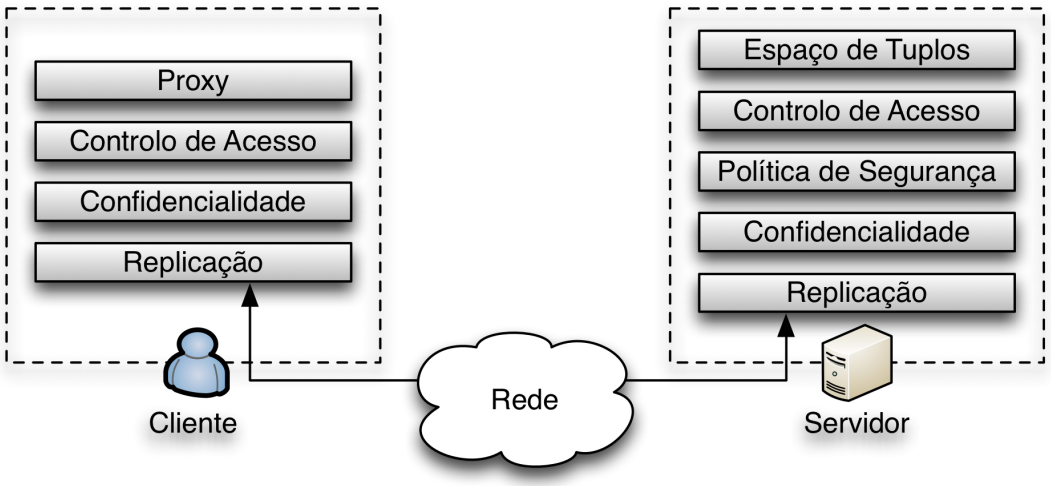

Figura 2.4: Arquitectura *DepSpace*

- *Policy Enforcement* - Para uma operação ser aprovada é necessário o *id* de quem invoca a operação e os argumentos dessa mesma operação. A verificação é local a cada servidor e o cliente só aceita a aprovação ou rejeição, se recebe *f+1* cópias do mesmo resultado [9]. Deste modo, toleramos comportamento arbitrário por parte de servidores faltosos, pois uma maioria de servidores correctos vão fazer uma verificação correcta da política. Esta política é definida pelo administrador aquando da configuração do sistema;

- Controlo de acesso - Assume-se que cada cliente tem o seu *id* único e que é adquirido pelos servidores quando é estabelecida uma ligação segura. Quando um espaço de tuplos é criado, o administrador cria também uma lista de controlo de acesso (*ACL*) com os *ids* dos clientes que podem inserir tuplos. Quando um tuplo é inserido, leva duas listas que definem quais os clientes que podem ler e remover o respectivo tuplo.

Com esta descrição das camadas do *DepSpace* é possível afirmar que há um conjunto de propriedades que este serviço consegue garantir. Em primeiro lugar, a camada de replicação garante que todas as operações, realizadas por parte de um cliente correcto, são executadas de acordo com a especificação de linearidade [10] (todas as réplicas executam as mesmas transições de estado, devido ao seu determinismo e à difusão com ordem total). A segunda propriedade garante que é extremamente complicado um adversário conseguir descobrir o conteúdo de um tuplo devido ao esquema *PVSS* usado na camada de confidencialidade, desde que no máximo f servidores sejam faltosos. O *DepSpace* garante também, como terceira propriedade, que os danos que um cliente malicioso possa

fazer são recuperáveis e limitados. Quando um processo correcto lê um tuplo confidencial inválido, inserido por um cliente malicioso, a verificação de consistência falha e é invocada uma reparação. Esta reparação remove o tuplo do espaço e coloca o cliente numa lista negra, o que faz com as suas que operações seguintes sejam negadas. A última propriedade é que todas as operações num espaço de tuplos são executadas se e só se estão em acordo com a política definida para esse mesmo espaço. A camada política de segurança satisfaz esta condição aprovando ou negando a operação em cada réplica, que se forem correctas, o seu estado é equivalente, logo produzem o mesmo resultado.

Em suma, o *DepSpace* é um serviço de coordenação que oferece tolerância a faltas bizantinas e que garante a confidencialidade dos dados inseridos num espaço de tuplo. Não podemos também esquecer que este nos oferece uma abstracção e um primitiva que nos permitem implementar serviços importantes como *locks* ou eleição de líder.

2.4 Comparação de Serviços de Coordenação

A tabela 2.1 apresenta uma comparação dos serviços de coordenação analisados nas secções anteriores, que nos permite uma melhor comparação. Como podemos ver, estes serviços de coordenação oferecem diferentes formas de garantir sincronização.

O modelo para manutenção de consistência de *cache* do *ZooKeeper* (*Watch*) é muito mais relaxado que o do *Chubby*, o que aumenta o desempenho do primeiro. Este, permite a colocação de um *Watch* para se receber eventos de notificação quando há uma alteração, enquanto que o *Chubby* atrasa essa modificação até que todas as *caches* de todos os clientes sejam invalidadas.

Em relação à escalabilidade destes serviços, o *ZooKeeper* é um serviço escalável para leituras, uma vez que estas operações são efectuadas localmente numa réplica, logo, o serviço escala linearmente à medida que se adicionam mais réplicas. No caso do *Chubby*, mesmo que sejam adicionados servidores, o serviço não vai escalar. Tal deve-se ao facto de todas as operações serem satisfeitas pelo líder, pelo que se forem adicionados servidores, apenas se conseguem tolerar mais faltas por paragem. Em relação ao *DepSpace*, todos os servidores executam o mesmo conjunto de mensagens na mesma ordem, logo, este serviço não é escalável.

2.5 Serviços de coordenação utilizados

Para a primeira fase do projecto temos um conjunto de pré-requisitos que têm que ser garantidos pelo serviço de coordenação. É necessário usar um serviço de alta disponibilidade, com bons desempenhos, tolerante a faltas por paragem e, principalmente, pronto para produção já que vai ser utilizado como base de desenvolvimento de todo um sistema, como se verificará mais à frente.

O melhor serviço de coordenação estudado para dar resposta a todos estes requisitos é o *ZooKeeper*. É um serviço que já está em funcionamento, em vários sistemas conhecidos, nomeadamente no *Facebook* [12] e na *Yahoo!*. Em relação aos outros serviços estudados, o *Chubby*, desenvolvido pelo *Google*, não é *software* livre e o *Sinfonia* é apenas um protótipo da *HP*.

Para a segunda fase do projecto, as necessidades foram basicamente as mesmas, mas era necessário a tolerância a faltas bizantinas. Esta nova necessidade obriga ao aumento da complexidade do serviço de coordenação e, consequentemente, à diminuição do desempenho, mas nunca descurando da disponibilidade máxima. Sem dúvida que o serviço de coordenação escolhido foi o *DepSpace*, pois é o único serviço de coordenação, entre os estudados, capaz de tolerar faltas bizantinas.

No entanto, como podemos ver na tabela 2.1, este serviço apresenta algumas limitações, embora seja o melhor em termos de segurança. No capítlo 4 explicamos as alterações feitas para colmatar essas limitações.

2.6 Considerações Finais

Neste capítulo apresentámos um estudo sobre os serviços de coordenação. Foram apresentadas as principais abstracções de memória partilhada e primitivas de coordenação suportadas, bem como uma descrição de quatro exemplos de serviços: *ZooKeeper*, *Chubby*, *Sinfonia* e *DepSpace*.

Entre estes serviços, o *ZooKeeper* é mais abrangente, com melhor desempenho e pronto para ser usado em aplicações reais, tendo portanto sido escolhido como base para a concretização da camada de coordenação da arquitectura *ReD*.

O serviço de coordenação que oferece melhores garantias de segurança, nomeadamente confidencialidade e tolerância a faltas bizantinas, é o *DepSpace*. Estes motivos levaram a que este serviço fosse escolhido para a concretização da coordenação tolerante a faltas bizantinas.

Seguidamente vamos demonstrar como é que estes serviços de coordenação escolhidos solucionam os desafios deste projecto.

Tabela 2.1: Comparação de Serviços de Coordenação

	ZooKeeper	*Chubby*	*DepSpace*	*Sinfonia*
Abstracção	Espaço de nomes hierárquico	Espaço de nomes hierárquico	Espaço de tuplos	Registos partilhados
Primitiva de Coordenação	Contador atómico	*Locks*	Extensão ao espaço de tuplos (*CAS*)	Minitransacções
Persistência	Escrita em disco antes de aplicada em memória	*Backups* para um *Google File System*	NA	Registos guardados em disco
Cache	Mecanismo de notificação baseado em eventos que permite a implementação (Watch)	Mecanismo de notificação baseado em eventos que permite a implementação	NA	Mecanismo de minitransacções permite a implementação
Tolerância a Faltas	Tolerância de faltas por paragem (*2f+1*)	Tolerância de faltas por paragem (*2f+1*)	Tolerância de faltas bizantinas (*3f+1*)	Depende da configuração, mas não tolera faltas bizantinas
Detecção de falhas	*HeartBeats* e *Timeout*	*KeepAlives* e *Leases*	NA	Baseada em sincronia
Escalabilidade	Sim	Não	Não	Depende da configuração, mas minitransacções escalam bem
Segurança	*ACL* e comunicação processo/ serviço segura	*ACL*	*ACL*, comunicação processo/ serviço segura e tolerância a intrusões	NA

Capítulo 3

Coordenação na arquitectura *ReD*

Neste capítulo vamos ilustrar a utilidade de um serviço de coordenação através do seu uso num problema real em sistemas distribuídos. Como foi referido na secção 2.5, o *ZooKeeper* foi o serviço de coordenação escolhido para a implementação de coordenação no projecto *ReD*.

Nas secções seguintes é dada uma explicação da arquitectura *ReD* para um melhor entendimento de onde podemos encaixar um serviço de coordenação num sistema distribuído complexo para facilitar o seu desenvolvimento.

Especificamos a camada de coordenação propriamente dita, ou seja, o que o serviço de coordenação tem de garantir para que o sistema tenha bases sólidas para ser desenvolvido. Descrevemos, também, um algoritmo que consegue garantir as propriedades necessárias e fazemos uma prova informal. Este algoritmo foi implementado e está em funcionamento, pelo que deixamos alguns detalhes levados em consideração nessa mesma implementação.

3.1 Arquitectura *ReD*

A figura 3.1 mostra uma visão geral da arquitectura *ReD*. Esta arquitectura é composta por três componentes de sistema: as réplicas do serviço, um sistema de armazenamento confiável (e.g. *SAN* [30]) e o serviço de coordenação.

Apenas a réplica considerada líder pode escrever no sistema de armazenamento. A comunicação entre todos os componentes do sistema vai ser explicada de seguida.

Os vários componentes de sistema usam protocolos de comunicação diferentes, tal como mostra a figura 3.2. Nesta figura, as setas pretas são indicativas do início das ligações entre dispositivos, uma vez que, após essas ligações estarem estabelecidas, a comunicação é feita bidireccionalmente.

Os clientes ligam-se a um *middleware*, que faz chegar os pedidos a todas as réplicas do serviço. A comunicação entre os clientes e o *middleware* e, ainda, entre este e as réplicas é feita através dos protocolos nativos de comunicação com sistemas gestores de base de

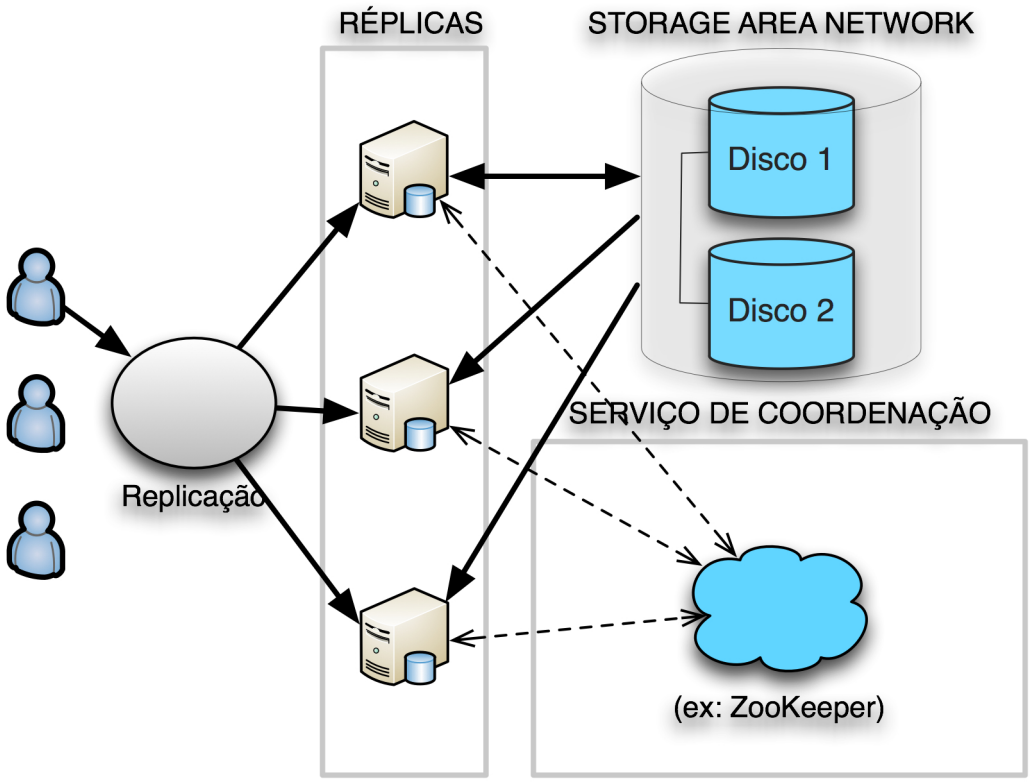

Figura 3.1: Arquitectura *ReD*

dados [11].

Antes dos pedidos chegarem propriamente às réplicas do serviço (servidores *MySQL*) são interceptados ao nível dos pedidos I/O pelas máquinas virtuais através do protocolo *blktap* [50]. Estes pedidos têm de ser interceptados pois as máquinas virtuais não têm de facto uma cópia da base de dados no seu disco. De seguida, este protocolo faz então chegar os pedidos às réplicas.

As réplicas comunicam directamente entre si, através do protocolo de estabilidade. Este protocolo serve para definir se os dados da *cache* local das réplicas podem ser usados ou não.

Cada réplica pode, então, ligar-se a um processo coordenador. A comunicação entre réplicas e coordenadores é garantida pelo protocolo de coordenação. Este protocolo garante que existe apenas uma réplica escritora, mesmo na presença de falhas, e assegura a reconfiguração do sistema quando tal é necessário através de um serviço de coordenação.

Por sua vez, um coordenador liga-se a um servidor *ZooKeeper*, replicado ou não, que irá tratar da coordenação propriamente dita do sistema.

Os coordenadores comunicam, então, com os servidores *ZooKeeper* através do protocolo inerente a este serviço, como já descrito na secção 2.3.3. Os servidores em conjunto com os coordenadores agem como uma única máquina de estados global tolerante a faltas.

O foco deste projecto incide sobre toda a camada de coordenação e na construção do

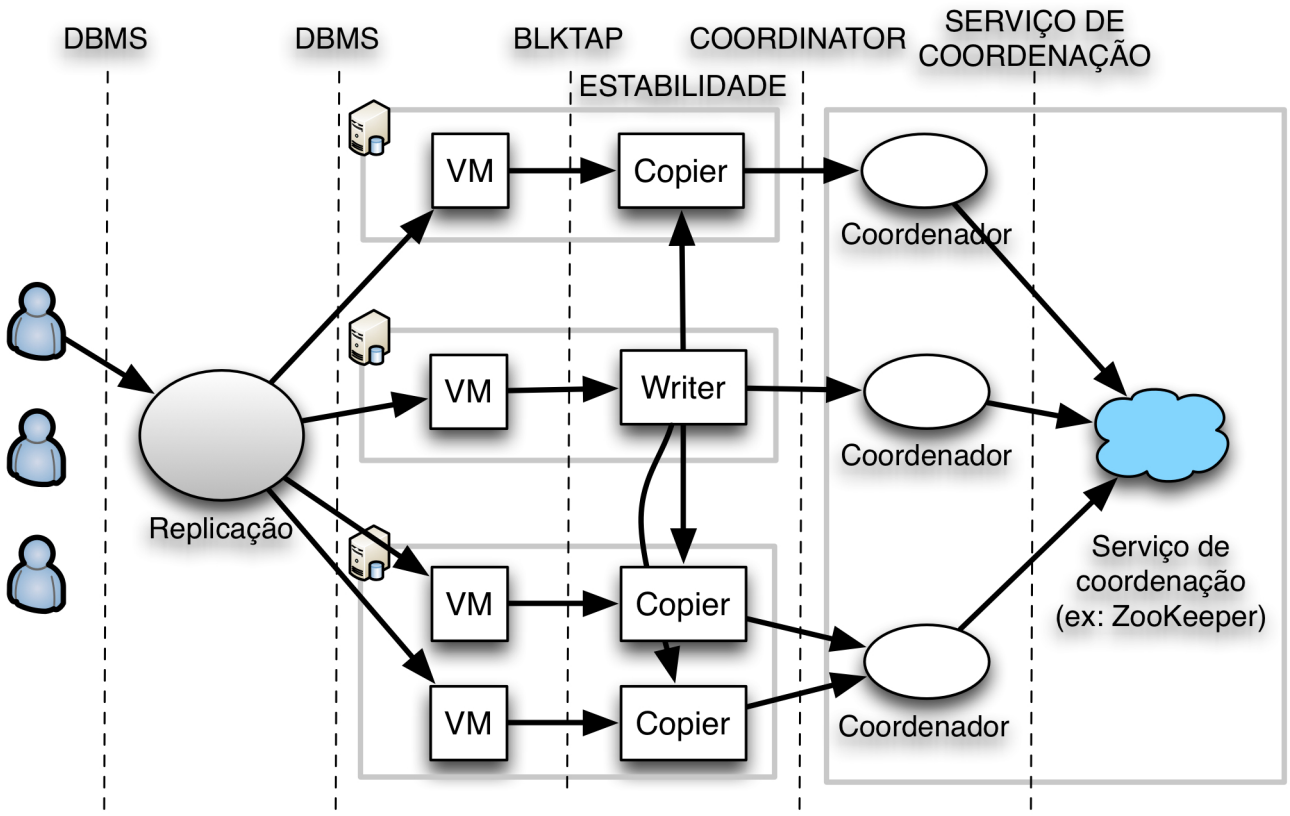

Figura 3.2: Protocolos do projecto *ReD*

algoritmo para o coordenador.

3.2 Especificação da camada de coordenação

Como foi dito anteriormente, o protocolo de coordenação da arquitectura *ReD* suporta uma *interface* bem definida e satisfaz um conjunto de propriedades. Esse conjunto é especificado nesta secção.

Nos casos de uso presentes podemos ver o nome de cada participante e o estado em que se encontra. O nome "coordenador" representa toda a camada de coordenação do sistema. Uma réplica pode ser *stopped* (uma nova réplica, acabada de iniciar comunicação), *copier* (uma réplica secundária) ou *writer* (a réplica líder).

Cada mensagem trocada entre a camada de coordenação e as réplicas é uma linha de texto composta por uma palavra-chave e uma lista de endereços (esta lista é opcional para algumas mensagens) Uma réplica envia e recebe mensagens apenas do seu coordenador correspondente. Existem as seguintes mensagens:

- *BOOTED id* - Uma nova réplica envia uma mensagem deste tipo, acompanhada pelo seu *id* aquando do início da comunicação. Neste ponto, a réplica deve ser considerada *stopped*;

- *ACK* - Esta mensagem é enviada pela réplica *writer* quando reconhece que existe uma nova réplica *copier*;

- *MAKECOPIER* - Uma réplica que receba esta mensagem sabe que a partir desse momento é considerada uma réplica *copier*;

- *MAKEWRITER id1,..., id3* - Quando uma réplica recebe uma mensagem deste tipo, percebe que a partir desse momento é a nova *writer* e recebe os *ids* das réplicas *copier*.

- *FAILED id1* - Esta mensagem é enviada pelo coordenador para a réplica *writer*, indicando que a réplica identificada por *id1* falhou.

A forma como estas mensagens são trocadas é ilustrada de seguida.

Na figura 3.3 ilustramos o caso em que uma réplica inicia. Se não existe nenhuma réplica *writer*, a nova réplica deve automaticamente ser considerada *copier* pelo coordenador.

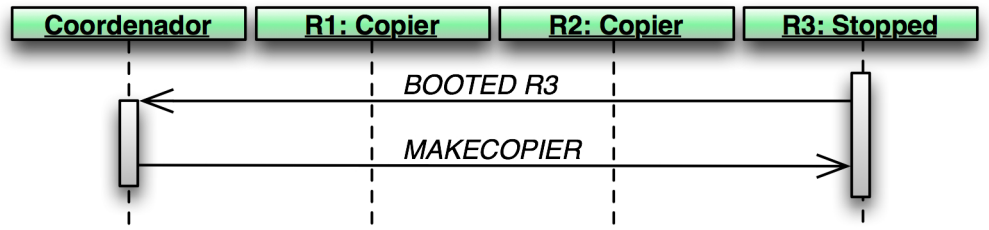

Figura 3.3: *ReD* (caso de uso 1) - Nova réplica inicia sem que exista uma *writer*.

Pode acontecer que o coordenador descubra que não existe uma réplica *writer*, embora existam várias *copiers*. Esta situação apenas se pode verificar ao iniciar o sistema ou quando a réplica *writer* falha. Posto isto, o coordenador deve escolher uma das réplicas *copier* para ser *writer*, enviando de seguida a mensagem "*MAKEWRITER R1,...,Rn*" (em que *R1,...,Rn* são os *ids* das réplicas *copier*) à réplica escolhida como mostra a figura 3.4. Não é necessário notificar directamente as *copiers* que existe uma nova *writer* nem esperar uma resposta por parte desta.

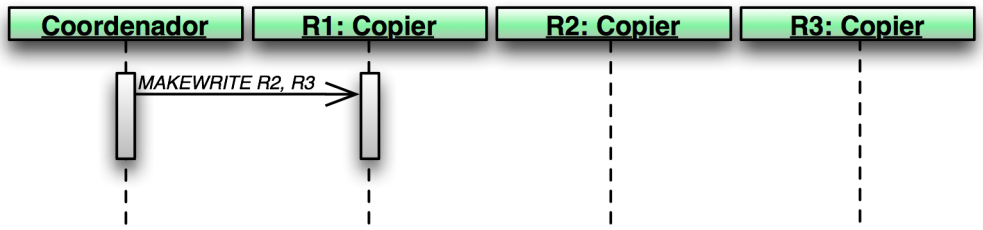

Figura 3.4: *ReD* (caso de uso 2) - Designando um novo *writer*.

Como podemos ver na figura 3.5, se uma réplica inicia quando existe uma *writer*, só pode ser transformada em *copier* quando a *writer* for notificada e a tiver reconhecido.

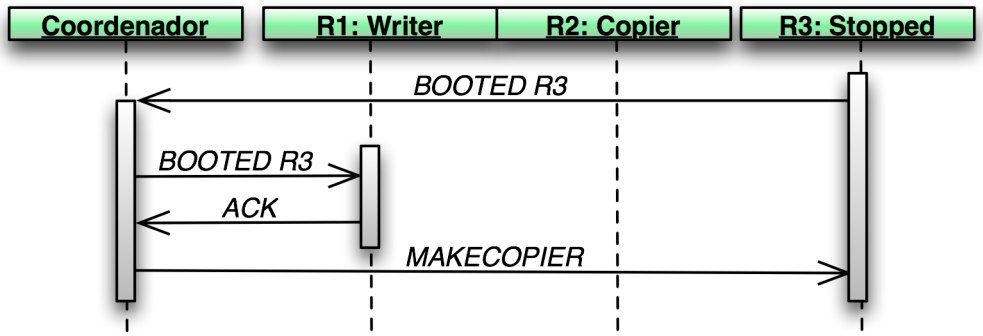

Figura 3.5: *ReD* (caso de uso 3) - Nova réplica *copier* com uma *writer* activa.

Em termos de faltas, assume-se que o coordenador consegue detectar réplicas faltosas, sem falsas suspeitas. Isto é equivalente a um detector de falhas perfeito [16] e é uma hipótese bastante aceitável em redes locais.

Uma réplica *writer* é notificada pelo coordenador quando alguma *copier* falha, deixando assim de a considerar *copier*, tal como mostra a figura 3.6.

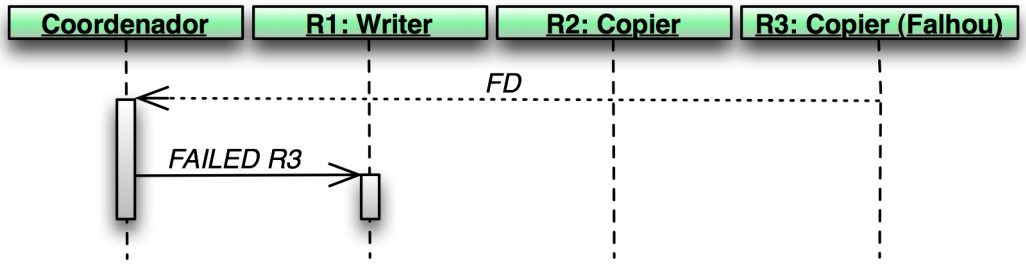

Figura 3.6: *ReD* (caso de uso 4) - Uma réplica *copier* falha quando existe uma *writer*.

No entanto, se uma réplica *copier* falha quando não existe uma *writer* activa (como no caso da figura 3.7), não é necessária nenhuma acção por parte do coordenador que seja mais do que esquecer essa réplica.

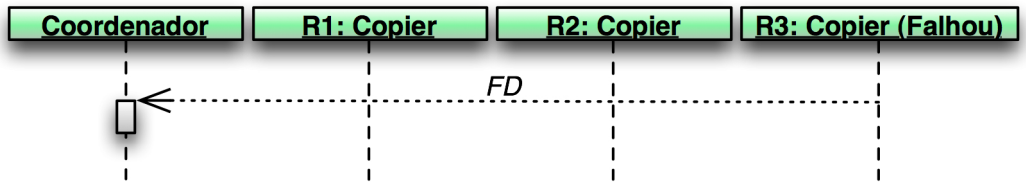

Figura 3.7: *ReD* (caso de uso 5) - Uma réplica *copier* falha quando não existe uma *writer*.

A figura 3.8 mostra-nos que quando a réplica *writer* falha e não existem pedidos pendentes, não é necessária nenhuma acção especial. No entanto, se existem réplicas *copiers* activas, é necessário garantir o caso da figura 3.4.

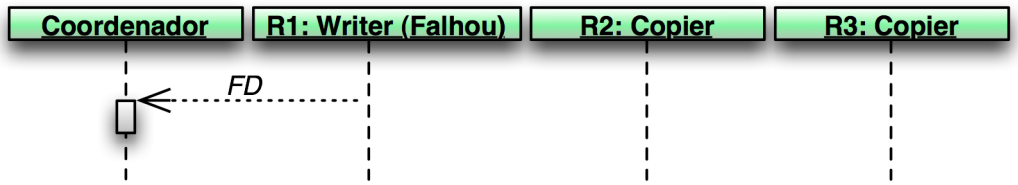

Figura 3.8: *ReD* (caso de uso 6) - Réplica *writer* falha.

Na figura 3.9 podemos ver que a réplica *writer* falha enquanto existem pedidos para tratar réplicas *stopped*. Num destes casos, os pedidos têm de ser tratados antes da nova réplica *writer* se anunciar a todas as *copiers*, como ilustrado na figura 3.4.

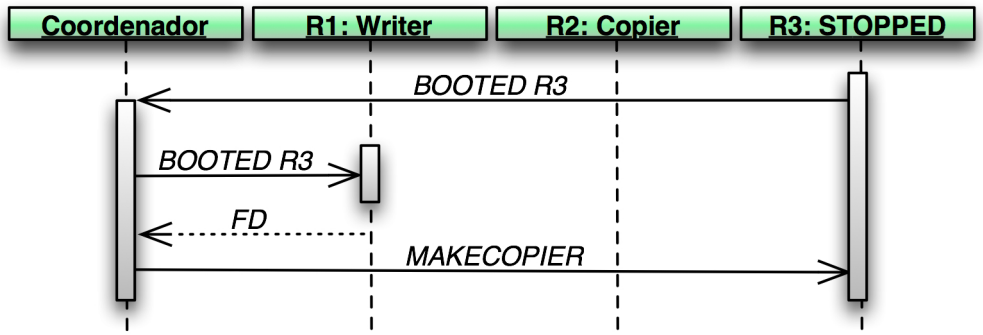

Figura 3.9: *ReD* (caso de uso 7) - Réplica *writer* falha antes de tratar de uma nova réplica a entrar no sistema.

3.2.1 Propriedades de correcção

A especificação descrita na secção anterior deve garantir certas propriedades de correcção. Estas propriedades dividem-se em condições *liveness* e *safety*.

O coordenador deve garantir as seguintes condições de *safety*:

S1: Existe no máximo uma réplica que se considera *writer*;

S2: Uma réplica pode considerar-se *copier*, se a *writer* o permitiu ou se nenhuma réplica se considera *writer*.

Em relação a *liveness*, o coordenador tem de garantir que:

L1: A menos que não existam *copiers*, acabará por ser designada uma *writer*;

L2: Quando a *writer* falha, o coordenador acaba por deixar de a considerar *writer*;

L3: Quando uma *copier* falha, a *writer* acaba por deixar de a considerar *copier*.

3.3 Solução para faltas *crash-stop*

Nesta secção apresentamos uma solução para a camada de coordenação baseada no *ZooKeeper* e tolerante a faltas *crash-stop*.

3.3.1 Modelo de sistema

O modelo de sistema assenta sobre um modelo *crash-stop* em que o serviço é replicado. Podem existir n_r réplicas e n_c coordenadores tal que $n_c \leq n_r$, onde até $n_c - 1$ coordenadores podem falhar. Assumimos também que quando uma réplica falha o coordenador coordespondente também falha, e vice-versa [1].

Existe um serviço de coordenação confiável que nunca falha e detecta perfeitamente falhas de coordenadores. Este serviço oferece as seguintes operações:

- *createNode()* - Permite criar um novo *znode* na hierárquia de nomes do *ZooKeeper*. Recebe como argumentos o nome a dar ao nó que vai ser criado, as opções de criação e o conteúdo do novo nó. As opções de criação podem ser sequencial e/ou efémero;

- *watch()* - Coloca um *watch* no *znode* passado em argumento. Pode também ser usado para colocar um *watch* em todos os nós de um nó pai;

- *getChildren()* - Devolve todos os nós filhos de um dado nó pai;

- *writeNode()* - Permite escrever no conteúdo de um nó. Recebe como argumentos o nome do nó no qual se quer escrever e o novo conteúdo;

[1] Na prática não é necessário que seja assim, visto que a réplica e o coordenador podem estar em máquinas físicas diferentes

3.3.2 Algoritmo

Para garantir as propriedades indicadas na secção anterior, desenvolvemos o algoritmo 1. De seguida, é feita uma análise detalhada a este algorimo, para uma melhor percepção do mesmo.

O método *onStart()* é chamado, quando uma réplica inicia uma ligação com o seu coordenador. Se esta réplica for a primeira fazer essa ligação, o coordenador cria o nó "/replicas" no *ZooKeeper*, servindo de nó pai para todos os nós das réplicas (não ilustrado no algoritmo). É criado exactamente um nó com as opções sequencial e efémero por cada uma das réplicas. A opção sequencial acrescenta ao nome do nó um número de sequência. Neste caso, o nome do nó é o *id* da nova réplica. A opção efémero serve para o serviço de coordenação saber que esse nó só existe enquanto houver uma ligação do coordenador com o servidor *ZooKeeper*. Este nó serve de proposta para o algoritmo de eleição de líder (*writer*). De seguida são colocados *Watches* em cada um dos nós filhos que já existem e um no nó pai "/replicas" (linhas 3 e 5). O objectivo é receber as notificações sobre as mudanças no grupo de réplicas. A operação *getChildren()* permite receber a lista de nós filhos para procuramos, então, se já existe um nó com o conteúdo "*W*", indicando que já existe uma réplica (*writer*). Em caso negativo, a réplica pode automaticamente assumir-se *copier* e o coordenador trata de escrever "*C*" (*copier*) no nó. De seguida é iniciado o processo de eleição da nova réplica (*writer*). A mensagem "*MAKECOPIER*" é enviada para a réplica mais tarde (ver *onNodeModification*). Se a nova réplica *writer* for a que iniciou, o coordenador escreve "*W*" (*writer*) no nó correspondente. Se já existir um *writer* na lista que foi pedida, o coordenador aguarda a escrita de "*C*" por parte do coordenador da réplica *writer*.

Quando algum nó é criado ou apagado de "/replicas", é gerado um evento *NodeChildrenChanged* e os coordenadores executam, respectivamente, *onNodeCreation()* ou *onFailure()*.

(1.) Caso um nó tenha sido criado, todos os outros coordenadores chamam o método *onNodeCreation()*, ou seja, caso uma nova réplica se ligue a um coordenador. Este método coloca um *Watch* no novo nó para que os coordenadores possam receber notificações caso aconteçam alterações (linha 12). O coordenador correspondente à réplica *writer*, envia-lhe "*BOOTED id*" e espera um "*ACK*" para que possa transformar a nova réplica *stopped* em *copier*, ou seja, para que possa escrever "*C*" (*copier*) no novo nó. Posteriormente, o evento *NodeDataChanged* vai ser gerado e o coordenador correspondente vai enviar "*MAKECOPIER*" à réplica (ver *onNodeModification*);

(2.) No caso de um nó ter sido apagado (i. e., a ligação entre o coordenador e a réplica foi perdida) o método chamado é o *onFailure()*. Para ratificar que existe um uma réplica *writer*, todos os coordenadores executam o processo de eleição. Esta execução é feita pois a *writer* pode ter falhado entretanto. De seguida, o coordenador pode então enviar a mensagem "*FAILED id*" à replica *writer*.

Algoritmo 1: Algoritmo de Coordenação *ReD Fail-Stop*

```
   onStart(C)
1    stopped←true;
2    createNode(C, SEQ, EPHEMERAL, "S");
3    watch(replicas);
4    R←getChildren(replicas);
5    ∀ ZNode Z ∈ R : watch(Z);
6    if ("W ∉ R) then
7        writeNode(C, "C");
8        if ("C" é o menor id em R) then
9            writerNode(C, "W");
10   else
11       writer = "W" em R;

   onNodeCreation(id) //NodeCreation, NodeChildrenChanged
12   watch(id);
13   if (C=writer) then
14       send "BOOTED id" to replica;
15       wait for "ACK" from replica;
16       writeNode(id, "C");

   onNodeModification(id, VALUE) //NodeModification
17   if (id = C && stopped && VALUE = "C") then
18       send "MAKECOPIER C" to replica;
19       stopped←false;
20   else
21       if (VALUE = "W") then
22           writer←id;
23           if (id = C) then // Im the new leader
24               R←getChildren(replicas);
25               ∀ ZNode ⟨id, "S"⟩ ∈ R : writeNode(id, "C");
26               R←R/{C};
27               send "MAKEWRITER R" to replica;

   onFailure(id) //NodeChildrenChanged
28   R←getChildren(replicas);
29   if ("C" é o menor id em R) then // Novo writer?
30       writeNode(C, "W");
31   if (writer = C) then
32       send "FAILED id" to replica;
```

O bloco *onNodeModification()* trata as alterações feitas aos nós existentes em "/replicas". Quando um desses nós é alterado, o evento *NodeDataChanged* é gerado e todos os coordenadores vão ser notificados, uma vez que todos colocaram um *Watch* em cada um dos nós. Se o valor "*C*" for escrito nalgum nó, o coordenador da réplica correspondente envia-lhe "*MAKECOPIER*". Se o valor escrito for "*W*", significa que existe uma nova réplica *writer* e o nome desse nó é guardado. Por último, se uma nova réplica writer foi eleita, o coordenador correspondente procura se existem nós com o valor "*S*" (*stopped*). Em caso afirmativo o coordenador escreve "*C*" nesses nós e envia "*MAKEWRITER*" às réplicas correspondentes (linhas 23-27). Esta procura de nós *stopped* deve-se ao facto do coordenador ter de tratar casos pendentes antes de poder anunciar à réplica que é a nova *writer* (ver figura 3.9).

De seguida, prova-se, informalmente, que este algoritmo garante as propriedades de *safety* e de *liveness* necessárias à camada de coordenação do *ReD*.

3.3.3 Prova Informal

Para se provar que o algoritmo garante as propriedades necessárias, faz-se referência às propriedades *safety* e *liveness* indicadas na secção 3.2, sendo explicadas por ordem.

S1: Para provar que existe no máximo uma réplica *writer*, temos que garantir que se não existir nenhuma, o processo de eleição vai ser iniciado e que se já existir alguma, mais nenhuma se considera também *writer*. O método *onStart()* garante que uma nova réplica vai procurar se existe algum nó com o valor "*W*". Em caso afirmativo, esta nova réplica sabe que existe uma réplica *writer*, pelo que não irá considerar-se também *writer*. Em caso negativo, é iniciado o processo de eleição em que a réplica que criou o *znode* com o número de sequência mais baixo sabe que é a nova *writer*. Esse número garante então que mais nenhuma réplica se considera *writer*. No entanto, uma réplica *writer* pode falhar e neste caso o evento *NodeChildrenChanged* é disparado e tratado no método *onFailure()*. Este método executa sempre o processo de eleição, quer seja pela falha efectiva da réplica *writer* ou pela ratificação por parte de todas as réplicas de qual é a *writer* existente (no caso de não ser a *writer a falhar*).

S2: A segunda propriedade de *safety* diz que uma réplica só se pode considerar *copier* se a *writer* o permitiu ou se nenhuma se considera *writer*. Como foi referido na propriedade anterior, uma réplica procura um nó com o valor "*W*" quando inicia para saber se existe ou não uma *writer*. Se a *writer* existir, recebe um evento *onChildrenChanged* e executa o método *onNodeCreation()*. Neste método, o coordenador correspondente envia a mensagem "*BOOTED id*" (com o *id* da nova réplica) à replica *writer*, espera uma resposta com mensagem "*ACK*" e escreve o valor "*C*" no nó da nova réplica. Quando o coordenador terminar essa escrita, todos os coordenadores recebem uma notificação do

evento *NodeModification*, tratado no método *onNodeModification()*. Neste método, o coordenador correspondente percebe que a *writer* já reconheceu a nova réplica, pelo que trata de enviar-lhe "*MAKECOPIER*" para que esta se passe a considerar *copier*.

L1: Em relação às propriedades *liveness*, a primeira diz que a menos que não existam *copiers*, eventualmente irá existir uma réplica *writer*. Só existem dois cenários em que não existe uma réplica *writer*. O primeiro é quando nunca existiu nenhuma réplica de qualquer tipo. Neste cenário, a primeira réplica que iniciar a ligação vai procurar se existe um *znode* com o valor "*W*", mas não vai existir nenhum. Sendo assim, essa réplica pode automaticamente considerar-se *copier* (ver figura 3.4) e executa o processo de eleição. Como o único *znode* existente é o dessa réplica, esta é a nova *writer*. O segundo cenário é quando a réplica *writer* existente falha. Neste caso os coordenadores das *copiers* recebem, eventualmente (devido a assincronia), uma notificação do evento *NodeChildrenChanged*. Desta vez, este evento é tratado no método *onFailure()* que inicia o processo de eleição de uma nova réplica *writer*. Sendo assim, conseguimos garantir que só não existe uma réplica que se considere *writer* se não existirem *copiers*. De notar que podem falhar *copiers* enquanto a nova *writer* não é eleita. Este facto é aceitável pela especificação, tal como podemos ver na figura 3.7.

L2: A segunda propriedade de *liveness* diz que quando a réplica *writer* falha, os coordenadores acabam por deixar de a considerar *writer*. Como foi dito na propriedade anterior, quando uma réplica falha é disparado o evento *NodeChildrenChanged* e tratado no método *onFailure()*. Este método inicia o processo de eleição e quando terminar, todos os coordenadores sabem quem é a nova réplica *writer*, deixando de considerar a antiga.

L3: Por último, a terceira propriedade de *liveness* diz-nos que quando uma réplica *copier* falha, a *writer* acaba por deixar de a considerar *copier*. Quando uma réplica falha é disparado o evento *NodeChildrenChanged*, neste caso tratado no método *onFailure()*. Este método garante que a *writer* deixa de considerar a réplica *copier* que falhou, pois o coordenador correspondente envia-lhe a mensagem "*FAILED id*" (em que "*id*" é o *id* da réplica que fahou).

Ficou, então, provado informalmente que o algoritmo garante as propriedades pedidas pelo sistema.

3.3.4 Detalhes de Implementação

O presente algoritmo foi implementado e integrado na arquitectura *ReD*. Deixamos, agora, um conjunto de detalhes relevantes à implementação de um sistema deste tipo.

Após uma extensa utilização das operações oferecidas pela *API* do *ZooKeeper*, percebemos que não se comportam todas da mesma forma no que toca a eventos disparados (se colocarmos um *Watch*).

***getChildren()*:** Este método recebe como argumento um *znode* pai para que sejam retornados os *znodes* filhos num objecto *List*. Cada elemento dessa lista contém uma *string* que é o *path* para cada *znode* filho, sendo que nenhuma ordem é especificada. De notar que não é o *path* completo (ex: /election/n_0000000000 é o *path* completo, mas o elemento da lista apenas contém n_0000000000). Esta observação permitiu-nos saber como dividir o nome dos *znodes* do número de sequência para poder ser usado na escolha de eleição de uma réplica *writer*. Se pedirmos ao método para colocar *Watches*, o evento *NodeChildrenChange* é disparado quando um *znode* filho é criado ou removido e nenhum evento é disparado se os dados de algum *znode* forem alterados (nem do *znode* pai, nem dos *znodes* filhos). Percebemos então que para sermos notificados de uma alteração aos dados de um *znode* era necessário a utilização da operação *getData()* que dispara *NodeDataChanged* (explicada de seguida). O evento *NodeDeleted* apenas acontece quando o *znode* pai é apagado, pelo que pode ser ignorado pelo nosso algoritmo. Para recebermos notificações sobre o estado dos nós filhos é necessário tratar o evento *NodeChildrenChange*.

***getData()*:** Nesta operação o evento *NodeDataChanged* é disparado quando os dados do *znode* são modificados. No entanto, também dispara o evento *NodeDeleted* quando o *znode* é removido e é ignorado pelo nosso algoritmo devido à razão apresentada na operação anterior.

***exists()*:** É uma operação útil para percebermos se já houve alguma réplica a criar o *znode* pai, pois retorna *true* ou *false* caso um dado *znode* já exista ou não. Após o uso desta operação podemos ignorar todas as excepções *KeeperException.NoNode* lançadas pelas outras operações, pois o nó pai existe sempre. Não usamos esta operação para mais nenhuma função, porque embora dispare muitos eventos úteis, não nos oferece uma visão mais geral como no caso da operação *getChildren()*.

Existem, também, alguns cuidados a ter quando se usa o sistema de notificação do *ZooKeeper*:

- Os *Watches* só disparam uma vez, pelo que se colocarmos um *Watch* num *znode*, apenas somos notificados uma vez sobre alguma alteração ao estado desse *znode*. Percebemos então que caso fosse necessário receber mais notificações sobre o mesmo *znode*, era preciso colocar um novo *Watch*;
- Os clientes *ZooKeeper* recebem as notificações, em tempos diferentes, mas sempre pela mesma ordem o que significa que conseguimos garantir determinismo, como

por exemplo, a eleição da mesma réplica *writer* por parte de todos os coordenadores. Se a réplica *writer* falhar, é disparado um evento que faz iniciar o processo de eleição, ou seja, todas as réplicas vão, mais cedo ou mais tarde, considerar a mesma réplica *writer* (após essa eleição).

- Um *znode* pode sofrer múltiplas alterações durante o intervalo de tempo em que um cliente recebe uma notificação e coloca outro *Watch* sobre o mesmo *znode*, logo, podem ser perdidos um ou mais eventos . Para resolver este problema, o primeiro passo a fazer quando algum evento dispara é a colocação de outro *Watch* do mesmo tipo.

3.4 Considerações finais

O protocolo de coordenação da arquitectura *ReD* é um componente fundamental para o bom funcionamento do sistema. Apesar de não ser um protocolo complexo, o algoritmo de coordenação apresentado neste capítulo tem muitas subtilezas que garantem que as propriedades de *safety* e *liveness* enunciadas na secção 3.2.1 sejam mantidas. Neste capítulo mostrámos que a solução implementada funciona e resolve o problema para uma aproximação *crash-stop*.

Na próxima secção apresentamos uma solução para coordenação tolerante a faltas bizantinas, utilizando o serviço de coordenação *DepSpace*.

Capítulo 4

Coordenação BFT

Nesta parte do projecto, propomos uma outra solução para a camada de coordenação da arquitectura *ReD*, tolerante a faltas bizantinas. Para esta nova solução, decidimos utilizar o serviço de coordenação *DepSpace* ao invés do *ZooKeeper*. No entanto, conforme foi mostrado no capítulo 2, o *DepSpace* tinha sérias limitações que tornavam impossível a implementação da tarefa proposta. Neste capítulo, descrevemos essas limitações e o trabalho que foi realizado a fim de as corrigir. Posteriormente, já com a ajuda das novas funcionalidades do *DepSpace*, mostramos como se consegue realizar uma solução tolerante a faltas bizantinas para a coordenação de réplicas na arquitectura *ReD*. Para além do algoritmo de coordenação, apresentamos uma prova informal para explicar que este garante as propriedades da especificação da camada de coordenação. Por fim, deixamos alguns detalhes importantes relativamente à implementação desse mesmo algoritmo.

4.1 Limitações do *DepSpace*

Como já foi referido em capítulos anteriores, o *DepSpace* original [8] apresenta algumas limitações que dificultam o seu uso na concretização da coordenação na arquitectura *ReD*. Nesta secção descrevemos essas limitações para elucidar sobre onde recaiu o ênfase na melhoria deste serviço de coordenação.

Replicação. A versão original do *DepSpace* estava implementada, tendo por base uma versão mais antiga do *BFT-SMaRt* [7]. Toda a parte da difusão com ordem total era realizada por essa biblioteca completamente integrada no *DepSpace*. Esta era uma das suas maiores limitações, uma vez que o código estava confuso e intrincado com o do *BFT-SMaRt*, perdendo-se um dos seus principais objectivos: a simplicidade.

Tempo. Outras das limitações do *DepSpace* era o facto deste não ter qualquer noção ou assumpção de tempo. Deste modo, tornava-se impossível qualquer tipo de mecanismo de detecção de faltas, pois sem qualquer tipo de evento ou *timeout* que ajude a perceber se

uma réplica está ou não em funcionamento, não é possível implementar uma camada de coordenação como a do capítulo anterior.

Manipulação de conjuntos de tuplos. O *DepSpace* não oferecia a possibildiade de manipular conjuntos de tuplos. Todas as operações eram referentes a um único tuplo de cada vez e, como iremos ver mais à frente, ler ou remover um conjunto de tuplos (segundo um *template*) é bastante útil.

4.2 *DepSpace 2*

Apresentamos então uma nova versão deste serviço de coordenação, o *DepSpace 2*. Nesta versão, foram implementadas extensões para colmatar as limitações referidas na secção anterior que vão ser explicadas de seguida.

Replicação. Como vimos, a versão original do *DepSpace* usava a biblioteca *BFT-SMaRt* apenas para difusão com ordem total. Para a nova versão deste serviço de coordenação usámos também uma nova versão dessa biblioteca, que oferece uma solução completa para replicação máquina de estados [42]. Toda a parte de comunicação entre as réplicas do serviço de coordenação é agora responsabilidade da biblioteca *BFT-SMaRt* (os clientes esperam $f + 1$ respostas iguais, em que f é o numero de faltas bizantinas toleradas).

Confidencialidade. Como vimos na secção 2.3.4, o *DepSpace* original oferecia a possibilidade de inserir tuplos confidenciais no espaço e essa possibilidade foi mantida na nova versão. O problema aqui em questão é que cada servidor envia a sua *share* do tuplo, para que o cliente junte $f+1$ *shares* diferentes para o conseguir reconstruir. No entanto, a nova versão da da biblioteca *BFT-SMaRt* faz uma comparação das respostas *byte* a *byte*, o que faz com que nunca nenhuma mensagem deste tipo seja entregue. No entanto, a biblioteca permite a implementação de classes "*Comparator*" e "*Extractor*" para que a reconstrução de tuplos seja feita correctamente, de acordo com a semântica da aplicação.

***rdAll* & *inAll*.** Como vimos, o *DepSpace* original não oferecia a possibilidade de manipulação de um conjunto de tuplos na mesma operação. No âmbito dos objectivos deste projecto, torna-se vantajoso um coordenador (cliente do serviço da camada de coordenação) ter noção do que está a acontecer em todo o espaço de tuplos. Por exemplo, se fosse necessário obter todos os tuplos existentes num dado espaço de tuplos, teria de se retornar um a um. Implementámos então as operações *rdAll* e *inAll* que permitem a um cliente ler ou remover um ou mais (potencialmente todos) os tuplos do espaço que correspondam com um *template*. Por exemplo, se o número de correspondências for 2, a lista retornada ao cliente contém no máximo dois tuplos que correspondam com o *template*

dado. No entanto, estas operações foram implementadas de forma a que se colocarmos 0 (zero) no número de correspondências, o conjunto retornado ao cliente contém todos os tuplos que correspondem com o *template* dado. De notar que a operação *inAll* remove, mas também retorna os tuplos.

RD & IN. Estas duas operações foram melhoradas em relação às da versão original do *DepSpace*. A operação *RD* serve para ler um tuplo que corresponda com um dado *template*, enquando que a *IN* para além de ler, também o remove. Estas operações diferem das operações normais *RDP* e *INP* porque são operações que bloqueiam enquanto não conseguirem retornar. Para conseguirmos implementar estas operações, usamos dois parâmetros de tempo. Um desses parâmetros encontra-se no ficheiro de configuração do *DepSpace* e consiste na definição do intervalo de tempo em que o cliente pergunta aos servidores se já existe algum tuplo que corresponda ao *template*, ou seja, a frequência da verificação. O outro parâmetro de tempo consiste no intervalo total em que a operação vai ficar bloqueada a fazer essas verificações. De notar que caso este segundo parâmetro seja 0 (zero), o cliente fica bloqueado por tempo ilimitado (ou seja, até que apareça um tuplo que corresponda ao *template*).

TimedTuples. Uma das maiores novidades do *DepSpace 2* são os *TimedTuples*. Este novo tipo de tuplos contém uma data de validade que, em conjunto com a nova operação *renew*, torna possível detectar falhas de clientes do serviço. Se um cliente criar um *TimedTuple* e não o renovar, o tuplo acaba por se tornar inválido e qualquer outro cliente que o tente ler percebe que o cliente que o criou está atrasado ou falhou. Esta validade é opcional, ou seja, é possível criar tuplos persistentes e tuplos com data de validade. A operação *renew* permite a renovação da validade de um *TimedTuple*, fazendo com que este não fique inválido. Caso a validade de um *TimedTuple* não seja renovada com esta operações, o tuplo acaba por se tornar inválido.

Para implementar esta nova funcionaldiade usámos um *timestamp* que a biblioteca *BFT-SMaRt* oferece sempre que entrega uma mensagem às réplicas. Este *timestamp* é bastante útil pois é igual em todas as réplicas, permitindo determinismo. No entanto, este *timestamp* só é entregue no caso das operações que alterem estado, ou seja, no caso de ser necessário um consenso entre as réplicas. Se uma operação for *read-only*, ou seja, não altera estado (como é o caso de uma leitura *RDP* e *RD*), este *timestamp* já não é oferecido. O problema é que estas operações precisam do *timestamp* para perceber se o tuplo que estão a ler está expirado, o que requereria a execução de consenso em todas as operações, afectando o desempenho de forma negativa. A solução encontrada para este problema foi uma solução híbrida. Quando uma operação de leitura encontra um tuplo, retorna-o ao cliente para este verificar a sua validade localmente e caso tenha expirado, efectua um pedido de leitura não *read-only*, ou seja, a biblioteca *BFT-SMaRt* fornece um *timestamp*

às réplicas para ser utilizado naquela operação. Com base nesse *timestamp* as réplicas tomam uma decisão determinista sobre a validade do tuplo. Se o tuplo tiver expirado, é removido e continua uma pesquisa por outro que corresponda ao *template* fornecido à operação de leitura. Todos os tuplos que correspondam e que não sejam válidos, serão removidos nesta pesquisa, que se terminar sem nenhuma correspondência válida, retorna *NULL*.

4.3 Coordenação tolerante a faltas bizantinas no *ReD*

Nesta secção apresentamos a nossa solução tolerante a faltas bizantinas para a camada de coordenação do *ReD*, utilizando o *DepSpace 2*. De notar que esta solução é referente apenas à camada de coordenação, explicada na secção 3.2, pelo que os outros componentes do sistema tinham também de ser reestruturados para que todo o sistema tolerasse este tipo de faltas.

4.3.1 Modelo de sistema

Tal como na secção 3.3.1, o modelo de sistema é *crash-stop* em que existem n_r réplicas e n_c coordenadores tal que $n_c \leq n_r$. Assumimos que existe um serviço de coordenação tolerante a faltas bizantinas capaz de detectar perfeitamente falhas por paragem nos n_c coordenadores e que quando uma réplica falha, um coordenador também falha (e vice-versa).

Este serviço de coordenação necessita de $3f + 1$ servidores em que f é o número de faltas bizantinas que consegue tolerar e oferece as seguintes operações:

- *out()* - Operação que permite inserir um novo tuplo no espaço;
- *rd()* - Operação que bloqueia a execução até conseguir ler um tuplo que combine com um dado *template*. Recebe também um parâmetro que define quanto tempo deve estar bloqueada (se o valor for zero, bloqueia por tempo ilimitado);
- *in()* - Remove um tuplo existente no espaço;
- *rdAll()* - Lê um conjunto de tuplos que combina com um dado *template*;
- *renew()* - Permite renovar o tempo de validade de um tuplo do espaço;
- *cas()* - Esta operação permite procurar se um tuplo existe no espaço (dando um *template*) e inserir um dado tuplo em caso negativo, atomicamente.

Devido à natureza completamente diferente do *DepSpace 2* quando comparado com o *ZooKeeper*, não assumimos que o serviço de coordenação detecta falhas perfeitamente,

mas sim que o sistema é síncrono [25], o que na prática acaba por ser equivalente na maioria dos sistemas.

4.3.2 Algoritmo

A camada de coordenação *BFT* no *ReD* é constituída pelas funções *onStart(*C*)*, *copier()*, *writer(*R*)* e *dealWithStopped(*R*)* tal como se pode ver no algoritmo 2. Assumimos que o espaço de tuplos a ser utilizado por todos os coordenadores já está criado pelo administrador de sistema. Todos os tuplos inseridos no espaço são *TimedTuples*.

Quando uma réplica inicia, é executado o método *onStart()*. Este método insere o tuplo <*REPLICA*, C, *STOPPED*> (em que C corresponde ao *id* da réplica) e executa um ciclo para detectar se a réplica *writer* criou o *COPIER*, necessário para a nova réplica avançar. Se existir de facto uma réplica *writer*, esta irá inserir o tuplo *COPIER*. Temos de considerar também uma situação em que a réplica *writer* falha antes de tratar do tuplo *STOPPED*. Se tal acontecer, o tuplo *WRITER* acaba por ficar inválido e é possível sair do ciclo. De seguida, testamos se foi ou não criado o tuplo *COPIER* necessário (linha 4) e em caso negativo podemos assumir que foi a *writer* que efectivamente falhou e a nova assume-se automaticamente *copier*. Depois disto o coordenador envia "*MAKECOPIER*" para a nova réplica.

A segunda parte do algoritmo consiste na função *copier()*. Esta função começa por fazer um ciclo que constitui a tentativa de uma réplica ser *writer*. Enquanto uma réplica *writer* for correcta, a validade do seu tuplo é constantemente renovada, logo, nenhuma outra réplica consegue ser eleita. Nesse ciclo, todos os coordenadores (com excepção do correspondente à réplica que já é *writer*) executam a operação *CAS* (linha 12), utilizando o *template* <*REPLICA*, $*$, *WRITER*>. Se nenhum tuplo corresponder a este *template*, uma réplica consegue inserir o seu tuplo *WRITER*.

Após conseguir introduzir o seu tuplo *WRITER*, a réplica passa a ser a *writer* (linha 13). O próximo passo é eliminar o tuplo anterior (<*REPLICA*, C, *COPIER*>) e executar o método *dealWithStopped(*R*)* (explicado no último parágrafo). De seguida, é então enviado "*MAKEWRITER* $R/\{C\}$" à réplica ($R/\{C\}$ corresponde aos *ids* de todas as réplicas, excepto o da *writer* que é a emissora).

Por fim, o coordenador entra num ciclo infinito que tem de executar enquanto for correcto, já no método *writer(*R*)*. O primeiro passo é renovar o tuplo *WRITER*, que foi criado para que todas as outras réplicas não consigam executar a operação *CAS*. Depois o coordenador compara uma nova lista dos tuplos com uma lista anterior (passada ao método como argumento) e se faltar algum tuplo na nova lista, significa que alguma réplica falhou e envia "*FAILED i*" para a *writer* para que esta perceba qual a réplica que falhou (linhas 21 a 24). Posteriormente, é passada uma lista de tuplos actualizada ao método *dealWithStopped(*R*)*.

O método *dealWithStopped(*R*)* recebe como argumento uma lista de tuplos. A ideia

Algoritmo 2: Algoritmo de Coordenação BFT

```
   onStart(C)
1    out(<REPLICA, C, STOPPED>);
2    while (rdp(<REPLICA, *, WRITER>) ≠ NULL && T = NULL) do
3        T←rd(<REPLICA, C, COPIER>, 1000);
4    if (T=NULL) then // replica writer falhou
5        out(<REPLICA, C, COPIER>);
6        in(<REPLICA, C, STOPPED>);
7    send "MAKECOPIER C" to replica;
8    copier();

   copier()
9    repeat
10       sleep a little bit;
11       renew(<REPLICA, C, COPIER>);
12       T←cas(<REPLICA, *, WRITER>,<REPLICA, C, WRITER>)
     until (T≠NULL); // a partir daqui já é writer;
13   in(<REPLICA, C, COPIER>);
14   R←rdAll(<REPLICA, *, *>);
15   dealWithStopped(R);
16   send "MAKEWRITER R/{C}" to replica; // enviar confirmação de writer
17   writer(R);

   writer(R)
18   while (true) do
19       renew(<REPLICA, C, WRITER>);
20       TEMP←rdAll(<REPLICA, *, *>);
21       if (TEMP.size() < R.size()) then // tuplo removido, replica falhou
22           forall (Tuple T ∈ R) do
23               if (T ∉ TEMP && T.match(<REPLICA, *, COPIER>)) then
24                   send "FAILED i" to replica;
25       R←TEMP;
26       dealWithStopped(R);
27       sleep a little bit;

   dealWithStopped(R)
28   forall (Tuple T ∈ R) do
29       if (T=<REPLICA, i, STOPPED> && <REPLICA, i, COPIER> ∉ R) then
30           send "BOOTED(i)" and wait for "ACK" from replica;
31           out(<REPLICA, i, COPIER>);
32           in(T);
```

é que o coordenador correspondente à nova réplica *writer* tem de tratar todos os tuplos *STOPPED* antes de lhe poder anunciar que é *writer* (linha 15). Para além disso, já depois de se considerar *writer* é necesário verificar constantemente se existem novos tuplos para que possam ser tratados (linha 26). Para estas duas situações, o método repete o processo para cada tuplo *STOPPED* encontrado: enviar de "*BOOTED* i" à réplica *writer*, esperar mensagem "*ACK*" como resposta, inserir o tuplo <*REPLICA*, i, *COPIER*>, remover o <*REPLICA*, i, *STOPPED*> e enviar "*MAKECOPIER*" à nova réplica.

De seguida, explicamos o objectivo da política de segurança do projecto e, posteriormente, provamos informalmente que este algoritmo garante as propriedades impostas pela especificação da camada de coordenação do *ReD*.

4.3.3 Política

Na camada *PolicyEnforcer* do *DepSpace* é possível fornecer uma política de segurança para ser executada pelo servidor [8]. Esta política permite restringir a acção de um cliente exactamente apenas ao que é especificado.

Como se toleram faltas bizantinas é necessário precaução contra coordenadores maliciosos. Assim, usamos a seguinte política:

- *out(<REPLICA, C, STOPPED>)* pode ser executado por um processo com um *id* C;
- *out(<REPLICA, C, COPIER>)* pode ser executado se <*REPLICA, C, STOPPED*> já existir no espaço de tuplos;
- *in(<REPLICA, C, STOPPED>)* pode ser executado se <*REPLICA, C, COPIER*> já existir no espaço de tuplos;
- *in(<REPLICA, C, COPIER>)* pode ser executado se <*REPLICA, C, WRITER*> já existir no espaço de tuplos;
- *renew(<REPLICA, C, *>)* pode ser executado por um processo com *id* C;
- *cas(<REPLICA, *, WRITER>,<REPLICA, C, WRITER>)* pode ser executado por um processo com *id* C;

Todas as outras operações não são autorizadas, concretizando assim uma política de segurança prudente.

4.3.4 Prova informal

Para provarmos que o algoritmo garante as propriedades, tal como no capítulo anterior, é necessário fazer referência às propriedades *safety* e *liveness* indicadas na secção 3.2. Mais uma vez, serão explicadas por ordem.

S1: Em relação às condições de *safety*, a primeira propriedade diz que existe no máximo uma réplica que se considera *writer*. Quando uma réplica se liga a um coordenador é criado um tuplo <*REPLICA*, C, *STOPPED*> (em que C corresponde ao *id* da réplica). Se não existir nenhum tuplo que contenha *WRITER*, a nova réplica pode automaticamente assumir-se *copier* (visto que não existe nenhuma réplica *writer*). Se esse tuplo já existir, o coordenador fica à espera que o tuplo <*REPLICA*, C, *COPIER*> seja criado pela réplica *writer*. Desta forma garantimos que no arranque nenhuma réplica se vai considerar *writer* se essa já existir. Depois de uma nova réplica avançar (*copier*) vai executar a operação *CAS* para tentar ser *writer*. Enquanto o tuplo *WRITER* não desaparecer, nenhum outro coordenador conseguirá executar essa operação com sucesso. Quando uma réplica *writer* falha, a validade do seu tuplo acaba por expirar e a primeira a conseguir realizar o *CAS* com sucesso é a eleita para nova réplica *writer*. Mais tarde é enviada a mensagem "*MAKEWRITER* $R/\{C\}$" à réplica para que esta saiba que é a nova *writer*.

S2: A segunda propriedade *safety* diz-nos que uma réplica só se pode considerar *copier* se a *writer* o permitiu ou se nenhuma se considera *writer*. Uma réplica que inicie executa o método *onStart(*C*)*. Este método executa um ciclo até que (1.) o tuplo <*REPLICA*, C, *COPIER*> exista ou (2.) o tuplo *WRITER* seja inválido. Desta forma, garantimos que enquanto o tuplo *WRITER* não for inválido, ou seja, enquanto existir uma réplica *writer*, a nova réplica vai esperar que o seu tuplo *COPIER* seja criado. Se a réplica *writer* não existe ou falhar, é possível sair do ciclo para que a nova réplica se considere automaticamente *copier*.

L1: Em termos das propriedades *liveness*, a primeira diz que a menos que não existam *copiers*, acabará por existir uma réplica *writer*. Garantimos esta propriedade pois assim que uma nova réplica se ligar percebe que não existe réplica *writer* (porque uma leitura ao tuplo *WRITER* devolve *NULL*) e assume-se automaticamente *copier*. Depois, no método *copier()* vai conseguir realizar com sucesso uma operação *CAS* (porque não existe nenhum tuplo *WRITER*) e inserir o seu próprio tuplo *WRITER*. Desta forma a réplica passa a considerar-se *writer*.

L2: A segunda propriedade *liveness* diz-nos que quando uma réplica *writer* falha, o coordenador acabará por deixar de a considerar *writer*. Se uma réplica falha, assumimos que o coordenador correspondente também falha. Em relação aos outros coordenadores, como a réplica *writer* falha, o tuplo *WRITER* não vai continuar a ser renovado, pelo que outra réplica acabará por ser *writer*.

? (Se a réplica *writer* falha, o coordenador continua a executar o ciclo do método *leader* até que seja necessário comunicar com ela. Quando isso acontecer, essa comunicação não vai ser possível. Sendo assim, o coordenador não continua a sua execução, pelo que

deixa de considerar a réplica *writer*) ?

L3: Finalmente, a terceira propriedade diz que quando uma *copier* falha, a réplica *writer* acabará por deixar de a considerar *copier*. No ciclo infinito uma réplica *writer* correcta executa pede a lista total de tuplos existentes no espaço (através da operação *rdAll()*) e por comparação a uma lista anterior percebe se uma ou mais *copiers* falharam. Para cada uma das réplicas que tenham falhado, o coordenador envia "*FAILED id*" à réplica *writer* (em que "*id*" corresponde ao *id* da réplica que falhou). Desta forma garantimos que a réplica *writer* deixa de considerar todas as *copiers* que tenham falhado.

Ficou, deste modo, provado informalmante que o algoritmo garante as novas propriedades pedidas pelas novas características do sistema.

4.3.5 Detalhes de Implementação

O algoritmo de coordenação apresentado neste capítulo foi implementado e está disponível para ser utilizado. Durante a sua implementação foram observados alguns detalhes que merecem o seu destaque nesta secção.

A primeira observação prende-se com a ordem de inserir e eliminar tuplos do espaço. Quando uma réplica inicia, insere um tuplo <*REPLICA, ID, STOPPED*> e quando passar a *copier*, o tuplo anterior tem de ser removido. Esta eliminação só pode ser feita se o tuplo *COPIER* já estiver inserido. O mesmo acontece se a réplica passar de *copier* para *writer*, ou seja, o tuplo *COPIER* só pode ser eliminado se o tuplo *WRITER* já estiver inserido. Isto é garantido pela política de segurança, como podemos ver na secção 4.3.3. Esta pequena nuance é crítica, no que diz respeito à lista de tuplos que o coordenador correspondente à réplica *writer* tem de monitorizar para perceber se algum tuplo foi inserido ou eliminado. Se um tuplo fosse eliminado antes de outro ser inserido, o coordenador poderia ter uma visão errada das réplicas existentes, o que não é aceitável para enviar mensagens de coordenação às réplicas. A solução passa então por inserir primeiro o tuplo e só depois remover o que já está desactualizado. Quando o coordenador executa a operação *rdAll()* para ter a visão das réplicas existentes, é feito um arranjo dessa lista para que apenas os tuplos mais actualizados sejam levados em conta. Este arranjo consiste em percorrer a lista de tuplos e perceber se já existe ou não uma versão mais actualizada de cada tuplo nessa lista. Para isso usamos uma lista auxiliar para adicionar apenas os tuplos que já sabemos que são os mais actualizados. Se este arranjo não fosse feito, o coordenador continuaria a ter uma visão errada caso pedisse a lista num momento em que um tuplo tinha sido inserido, mas que o desactualizado ainda não tinha sido removido. Depois desse arranjo conseguimos garantir, por exemplo, que o tamanho da lista é o número de réplicas existente.

Notámos também que é necessário executar um ciclo quando uma réplica inicia. Este

ciclo consiste em procurar pelo tuplo <*REPLICA, C, COPIER*> enquanto este não existir e o tuplo *WRITER* ainda for válido. É necessário um ciclo porque uma réplica *writer* pode falhar depois de verificarmos que o tuplo *WRITER* existe e antes de conseguir inserir o novo tuplo *COPIER*. Assim garantimos que uma nova réplica só avança caso a *writer* falhe ou caso esta insira o novo tuplo *COPIER*.

De notar também que todos os parâmetros temporais existentes no algoritmo foram ajustados às necessidades de verificação de testes e não às necessidades do serviço propriamente dito, pelo que é possível a alteração desses valores.

4.4 Considerações finais

Neste capítulo mostrámos que é possível desenvolver uma solução tolerante a faltas bizantinas para a camada de coordenação do *ReD*. Apresentámos um algoritmo e a sua prova informal, bem como detalhes da sua implementação com o serviço *DepSpace*. Este serviço foi melhorado para permitir essa implementação, pelo que as suas limitações e consequentes melhorias também foram explicadas.

Capítulo 5

Conclusão

Este relatório, enquadrado num Projecto em Engenharia Informática (PEI), contribuiu para a investigação de um importante e recente tópico: os serviços de coordenação e sua aplicação na construção de sistemas distribuídos complexos.

Estes serviços oferecem grandes vantagens para os programadores, quando bem utilizados. As potencialidades explicadas ao longo deste relatório fazem com que os serviços de coordenação sejam bastante bem recebidos por parte dos programadores [14] [29]. No entanto, não tinha sido encontrada, até à data, uma comparação de serviços de coordenação existentes.

Para além disso, analisámos a arquitectura *ReD* que como vimos combina uma solução híbrida (*shared-storage* e *shared-nothing*) para a replicação de bases de dados e é composta por três sistemas principais: as réplicas do serviço, um sistema de armazenamento confiável partilhado e um serviço de coordenação.

Este PEI, planeava duas tarefas principais, que foram concluídas com sucesso. Foram elas:

- A concretização da camada de coordenação enquadrada na arquitectura *ReD*, num modelo *crash-stop*. Para tal, foi utilizado o serviço de coordenação *ZooKeeper*, demonstrando as suas potencialidades práticas e aplicando a teoria estudada. Foi desenvolvido um algoritmo, explicado, provado e concretizado, para fazer face às propriedades exigidas pela especificação da camada, por forma a que a solução fosse enquadrada na arquitectura. Esta solução foi desenvolvida e está em funcionamento.

- A concretização da mesma camada de coordenação do ponto anterior, mas tolerante a faltas bizantinas. Nesta segunda tarefa, o serviço de coordenação utilizado foi o *DepSpace*. Explicámos que a versão original deste serviço tinha algumas limitações, pelo que foi desenvolvida uma segunda versão a fim de adicionar extensões e funcionalidades de modo a permitir a implementação da camada de coordenação do *ReD*. Da mesma forma, foi desenvolvido, explicado e provado um

algoritmo que satisfaz as propriedades exigidas pela especificação da camada e que tolera faltas bizantinas.

Este relatório será também útil para trabalho futuro, não só no que diz respeito ao desenvolvimento do *DepSpace*, mas também no que toca ao desenvolvimento de qualquer sistema distribuído que tenha por base um serviço de coordenação.

Relativamente ao planeamento definido no relatório preliminar houve um atraso de dois meses devido a: (1.) falta da especificação da camada de coordenação da arquitectura *ReD* no início do projecto, (2.) necessidade de intervenção com outros investigadores do projecto *ReD*, nomeadamente da Universidade do Minho e (3.) complicações na concretização do *DepSpace 2* e no uso da biblioteca *BFT-SMaRt* (durante o projecto foram identificadas limitações nesta biblioteca que tiveram de ser colmatadas por outros investigadores do *LASIGE*). No entanto, este tipo de problemas eram mais ou menos esperados já que a bolsa alocada para este PEI era de 12 meses.

5.1 Trabalhos Futuros

Deixamos, então, uma lista de tarefas que podem ser desenvolvidas no futuro:

- Avaliar e optimizar o *DepSpace 2*: Devido à falta da especificação da camada de coordenação da arquitectura *ReD*, não houve tempo suficiente para realizar testes de desempenho da nova versão do *DepSpace*. Pensamos ser uma mais valia a comparação, por exemplo, com o serviço de coordenação *ZooKeeper* e qual a diferença de desempenho devido à adição de tolerância a faltas bizantinas.

- Guardar tuplos no disco: O *DepSpace* guarda os tuplos apenas em memória, mas seria interessante se se pudesse, também, guardar duplos de uma forma persistente em disco. O desempenho seria afectado, mas seria também uma forma de permitir a comparação directa e justa com o serviço de coordenação *ZooKeeper*.

  Seria extremamente interessante verificar o desempenho do *DepSpace* após esta funcionalidade estar implementada.

- *ReD BFT*: Sem dúvida alguma, que o trabalho futuro passa por uma aproximação tolerante a faltas bizantinas para toda a arquitectura *ReD*. Neste projecto mostrámos que tal objectivo é possível pelo menos na camada de coordenação e seria aliciante concluir toda a arquitectura tolerante a faltas bizantinas.

Bibliografia

[1] Resilient databases. http://red.lsd.di.uminho.pt.

[2] Atul Adya, John Dunagan, and Alec Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *NSDI*, pages 1–16. USENIX Association, 2010.

[3] Marcos K. Aguilera, Christos Karamanolis, Arif Merchant, Mehul Shah, Alistair Veitch, Marcos K. Aguilera, Christos Karamanolis, Arif Merchant, Mehul Shah, and Alistair Veitch. Building distributed applications using sinfonia, 2006.

[4] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *SIGOPS Oper. Syst. Rev.*, 41:159–174, October 2007.

[5] Amazon. Amazon simple queueing service. http://docs.amazonwebservices.com/AWSSimpleQueueService/ latest/ SQSGettingStartedGuide/.

[6] Kenneth Barclay and John Savage. *Groovy Programming: An Introduction for Java Developers*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[7] Alysson Bessani and Paulo Sousa. BFT-SMaRt. http://code.google.com/p/bft-smart/.

[8] Alysson Neves Bessani, Eduardo Pelinson Alchieri, Miguel Correia, and Joni da Silva Fraga. Depspace: A byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Systems Conference - EuroSys 2008*, April 2008.

[9] Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. Sharing memory between byzantine processes using policy-enforced tuple spaces. *IEEE Trans. Parallel Distrib. Syst.*, 20:419–432, March 2009.

[10] Alysson Neves Bessani, Miguel Correia, Joni da Silva Fraga, and Lau Cheuk Lung. Decoupled quorum-based byzantine-resilient coordination in open distributed systems. In *IN PROCEEDINGS OF THE 6TH IEEE INTERNATIONAL SYMPOSIUM ON NETWORK COMPUTING AND APPLICATIONS*, pages 231–238, 2007.

[11] Tamira Bonar and James Driscoll. A very easy hierarchical dbms implementation. In *Proceedings of the 1979 annual conference*, ACM '79, pages 45–53, New York, NY, USA, 1979. ACM.

[12] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1071–1080, New York, NY, USA, 2011. ACM.

[13] Tony Bourke. *Server load balancing*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[14] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[15] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, November 2002.

[16] Tushar Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43:225–267, March 1996.

[17] Orlando de Andrade Figueiredo. Implementação de espaços de tuplas do tipo javaspaces., 2002.

[18] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19:624–633, November 1976.

[19] Apache Software Foundation. Zookeeper, 2011. http://zookeeper.apache.org/.

[20] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1st edition, 1999.

[21] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7:80–112, January 1985.

[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[23] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Oper. Syst. Rev.*, 23:202–210, November 1989.

[24] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[25] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Ithaca, NY, USA, 1994.

[26] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. In *Readings in database systems (3rd ed.)*, chapter Principles of transaction-oriented database recovery, pages 235–250. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[27] Todd Hoff. Zookeeper - a reliable scalable distributed coordination system, 2008. http://highscalability.com/blog/2008/7/15/zookeeper-a-reliable-scalable-distributed-coordination-syste.html.

[28] Mike Hogan. Shared-Disk vs. Shared-Nothing - Comparing Architectures for Clustered Databases.

[29] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[30] IBM. Introduction to storage area networks. http://www.redbooks.ibm.com/abstracts/sg245470.html?Open.

[31] IBM. Tspaces. http://www.almaden.ibm.com/cs/TSpaces/.

[32] Marinus Frans Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Centrale Huisdrukkerij Vrije Universiteit, Amsterdam, 1992.

[33] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.

[34] Jean-Philippe Martin and Lorenzo Alvisi. Fast byzantine consensus. *IEEE Trans. Dependable Secur. Comput.*, 3:202–215, July 2006.

[35] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2:181–197, August 1984.

[36] Hugo Miranda, Alexandre Pinto, and Luís Rodrigues. Appia communication framework. http://appia.di.fc.ul.pt/.

[37] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. *SIGOPS Oper. Syst. Rev.*, 19:40–52, April 1985.

[38] David Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27:18–26, January 1993.

[39] F. D. Muñoz, H. Decker, J. E. Armendariz, and J. R. Gonzalez. Database replication approaches. October 2007.

[40] Bruce Jay Nelson. *Remote procedure call*. PhD thesis, Pittsburgh, PA, USA, 1981. AAI8204168.

[41] Benjamin Reed and Flavio P. Junqueira. A simple totally ordered broadcast protocol. *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference*, pages 245–256, June 2011.

[42] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22:299–319, December 1990.

[43] Markus Stadler. Publicly verifiable secret sharing. In *Proceedings of the 15th annual international conference on Theory and application of cryptographic techniques*, EUROCRYPT'96, pages 190–199, Berlin, Heidelberg, 1996. Springer-Verlag.

[44] Michael Stonebraker. The case for shared nothing. *Database Engineering*, 9(1):4–9, 1986.

[45] Ben Stopford. Understanding the shared nothing architecture. http://www.benstopford.com/2009/11/24/understanding-the-shared-nothing-architecture/.

[46] Francisco Javier Thayer Fábrega and Joshua D. Guttman. Copy on write, 1995.

[47] Ronald C. Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Experiences with locking in a NUMA multiprocessor operating system kernel. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[48] Paulo Verissimo and Luis Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[49] George Wells. Coordination languages: Back to the future with linda. In *Proceedings of WCAT'05*, pages 87–98, 2005.

[50] Xen Wiki. blktap. http://wiki.xensource.com/xenwiki/blktap.